Transformation binaire de niveau de fonction dynamique axée sur les performances

Arif Ali Anapparakkal

To cite this version:

Arif Ali Anapparakkal. Transformation binaire de niveau de fonction dynamique axée sur les performances. Other [cs.OH]. Université de Rennes, 2019. English. NNT : 2019REN1S114 . tel-02972362

HAL Id: tel-02972362
https://theses.hal.science/tel-02972362v1

Submitted on 20 Oct 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *Informatique*

Par

Arif Ali ANAPPARAKKAL

Performance Centric
Dynamic Function Level Binary Transformation

Thèse présentée et soutenue à Rennes, le Dec 9, 2019
Unité de recherche : Institut National de Recherche en Informatique et Automatique (Inria)
Thèse N° :

Rapporteurs avant soutenance :
Karine HEYDEMANN, Maître de conférence, Sorbonne Université
Henri-Pierre CHARLES, Directeur de Recherche, CEA

Composition du Jury :
Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du Jury ne comprend que les membres présents

Président :	Prénom Nom	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Philippe CLAUSS	Professeur, Université de Strasbourg
	Sandrine BLAZY	Professeur, Université de Rennes 1
	Sébastien FAUCOU	Maître de conférence, Université de Nantes
Dir. de thèse :	Erven ROHOU	Directeur de recherche INRIA

Invité(s) :
Prénom Nom   Fonction et établissement d'exercice

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my thesis advisor, Erven Rohou, for giving this opportunity to work under him. I would like to thank him for his continuous support, patience and motivation during my study.

I am extremely thankful to the esteemed members of jury for having agreed to examine my work.

I would like to thank the members of ALF and PACAP teams for their help and support and also for making my stay at Rennes really enjoyable.

I thank Dr. Arjun Suresh for his continuous encouragement and help during my Bachlor's, Master's and Doctoral degree studies.

I would like to thank my parents for their continuous help and support through out my life. This thesis is dedicated to them. I thank my wife, daughter and other family members for their prayers and support.

Above all I humbly bow my head before the Almighty who blessed me with energy and enthusiasm to complete this endeavour successfully.

TABLE OF CONTENTS

Résumé 7

Introduction 15

1 Background 21

1.1 Traditional Optimization Techniques 21
1.2 Prior Works 26
1.2.1 Static Optimizations 27
1.2.2 Dynamic Optimizations 30
1.2.3 Proposed work 32
1.3 PADRONE 33

2 FITTCHOOSER: A Dynamic Feedback-Based Fittest Optimization Chooser 35

2.1 Introduction 35
2.2 Survival of the fittest 37
2.2.1 Profiling 37
2.2.2 Optimization Pass 38
2.2.3 Cruise Control 39
2.3 FITTCHOOSER 39
2.3.1 Padrone 41
2.3.2 Implementation 42
2.3.3 FITTLAUNCHER 45
2.4 Results 47
2.4.1 Overhead 47
2.4.2 Speedup 48
2.4.3 FITTLAUNCHER 51
2.5 Related Work 51
2.6 Conclusion 53

3 OFSPER: Online Function Specializer 55
3.1 Introduction 55
3.2 Function Specialization 56
3.3 Dynamic Function Specialization 57
3.3.1 Use case 58
3.3.2 Our Approach 58
3.4 Implementation Details 62
3.4.1 Overview 62
3.4.2 OFSPER 63
3.5 Example 68
3.6 Result 70
3.6.1 Experimental set-up 70
3.6.2 Overhead 70
3.6.3 Speedups 71
3.7 Related Work 72
3.8 Conclusion 74

4 Implementation Details 75
4.1 Monitor Function 75
4.2 LLVM Passes 77

5 Conclusion 81
5.1 Publications 82
5.2 Further Extension 82

A Monitor function for FITTCHOOSER 85

B Monitor function for OFSPER 88

Bibliography 91

List of Figures 101

List of Tables 103

Résumé

De par sa nature statique, un compilateur a une visibilité limitée pour tenir compte de l'environnement dynamique ou du comportement d'une application. Les inconnues incluent les données d'entrée réelles qui ont une incidence sur les valeurs transmises par le programme et les détails spécifiques au matériel. Les entrées d'un programme peuvent considérablement changer l'efficacité d'un code compilé statiquement, et de nombreux cas sont soumis à une incertitude totale. Normalement, un compilateur statique applique partiellement des techniques d'optimisation dépendantes des données en effectuant des estimations approximatives des données d'exécution, parfois à l'aide d'une session de profilage réalisée lors de la compilation statique. Par exemple des optimisations comme la restructuration de boucles, l'élimination de variables d'induction, le déroulage ou le déplacement d'invariant sont toutes appliquées avec l'hypothèse que la boucle s'exécute un nombre minimum de fois. Ces techniques et d'autres, telles que la vectorisation, le tuilage de boucles, l'élimination de code inaccessible, etc., pourraient tirer parti d'information dynamique.

Les fonctionnalités matérielles modernes peuvent améliorer les performances d'une application, mais les éditeurs de logiciels sont souvent limités au plus petit dénominateur commun afin de maintenir la compatibilité binaire avec tout le spectre de processeurs utilisés par leurs clients. Avec des informations plus détaillées sur les fonctionnalités matérielles, un compilateur peut générer un code plus efficace. Mais même si le modèle de processeur exact est connu, les fabricants ne divulguent pas tous les détails.

Une solution serait de compiler un code source après avoir obtenu toutes les informations sur le programme que les différentes optimisations requièrent. Nous savons qu'une telle solution est impraticable. Une solution presque optimale est d'optimiser au cours de l'exécution du programme. Le principal avantage de l'optimisation est que l'optimiseur a une connaissance complète de l'environnement d'exécution, y compris des détails spécifiques au matériel. L'optimiseur obtient ainsi les données d'entrée et

peut ainsi, dans de nombreux cas, prédire le comportement futur du programme.

Cette thèse propose deux outils indépendants, FITTCHOOSER et OFSPER, qui appliquent des optimisations dynamiques au niveau des fonctions. L'idée de base est de remplacer une fonction existante par une version optimisée de celle-ci. FITTCHOOSER, un sélecteur d'optimisation basé sur le retour d'information dynamique, aide à trouver une version optimisée d'une fonction en comparant les performances de différentes versions de la fonction créée de manière dynamique. OFSPER, spécialiseur de fonctions en ligne, spécialise dynamiquement une fonction en fonction des valeurs réelles de ses arguments.

FITTCHOOSER: Un sélecteur d'optimisation à rétroaction dynamique

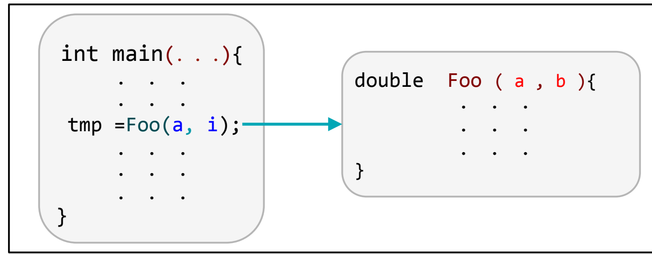
L'introduction de fonctionnalités avancées telles que les compteurs matériels et les unités de traitement vectoriel dans les microprocesseurs modernes peut permettre aux programmes de s'exécuter plus rapidement sans nécessiter de modification du code source. Les programmes peuvent même être optimisés au moment de l'exécution en surveillant de manière dynamique les compteurs matériels. Toutefois, les éditeurs de logiciels doivent toujours tenir compte de la compatibilité matérielle avant d'activer ces fonctionnalités avancées dans leurs applications. De nombreux modèles de processeurs ne prennent pas en charge les toutes dernières fonctionnalités d'optimisation et génèrent une erreur matérielle si un programme tente de les appeler. Mais même si le fournisseur pouvait compiler le programme directement sur chaque machine de déploiement séparément – ce qui est très peu pratique – les meilleurs compilateurs commerciaux et à code source ouvert manquent souvent des opportunités d'optimisation. Cela est dû en partie au manque d'informations publiques sur les détails de bas niveau des fonctionnalités du CPU. Sans modèle précis des caractéristiques de performance du processeur, le compilateur a recours à des méthodes heuristiques pour sélectionner des facteurs essentiels tels que le nombre de déroulements de boucles ou l'ordonnancement des instructions de chargement.

Cette thèse propose un outil d'optimisation dynamique appelé FITTCHOOSER pour surmonter ces limitations en générant des variations du code machine du programme

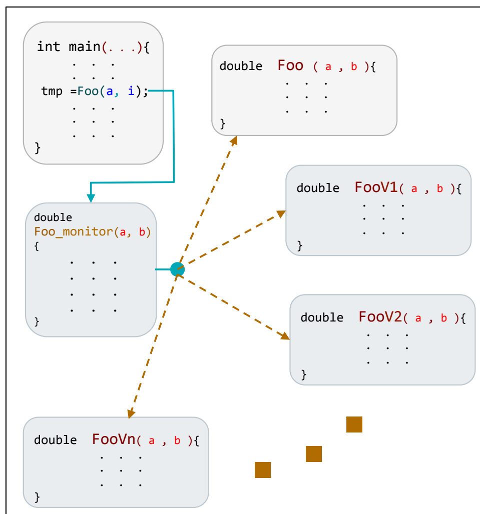
et en évaluant empiriquement les variations afin de sélectionner le plus performant. Ce processus itératif permet à FITTCHOOSER de trouver la technique d'optimisation la mieux adaptée aux fonctions les plus gourmandes en ressources d'un programme dans son environnement d'exécution actuel. Pour prendre en compte les changements potentiels dans les caractéristiques de performance, pouvant par exemple être causés par l'extension d'un tableau fréquemment traversé au-delà de la capacité du cache L3, FITTCHOOSER surveille en permanence ses fonctions optimisées et redémarre le processus d'évaluation lorsque des changements importants sont observés.

Les performances d'une exécution donnée d'un programme peuvent être affectées par un large éventail de facteurs, ce qui rend difficile de déterminer à l'avance quelles techniques d'optimisation sont les plus avantageuses. Bon nombre de ces facteurs peuvent être totalement imprévisibles. Par exemple, si le programme traite un flux d'entrée représentant l'activité de l'utilisateur final, il peut s'avérer impossible de prédéterminer les optimisations idéales pour une période donnée de ce flux d'entrée. Même si un compilateur choisissait les optimisations idéales pour un scénario d'exécution donné, le même programme compilé pourrait être exécuté dans un scénario légèrement différent, dans lequel d'autres optimisations amélioreraient les performances. Pour combler cet écart entre l'optimisation au moment de la compilation et l'exécution concrète d'un programme, FITTCHOOSER utilise une instrumentation dynamique pour générer et tester diverses combinaisons d'optimisations au début de l'exécution du programme, puis transformer le programme pour utiliser la combinaison qui s'avère empiriquement être la le plus efficace. Ceci est mis en œuvre sous forme de progression en trois phases:

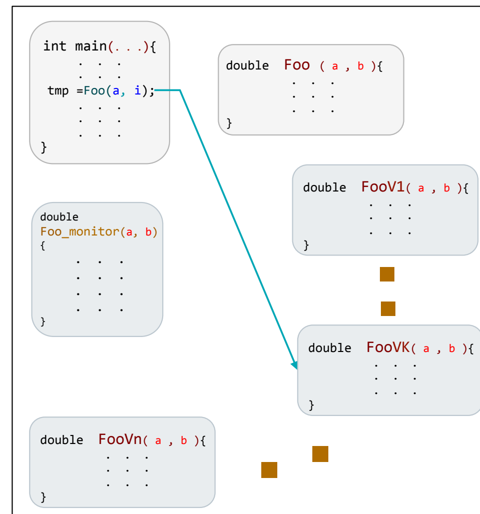
- *Profilage*: Identifier les cinq fonctions les plus gourmandes en ressources processeur de l'exécution en cours.
- *Passe d'optimisation*: Générer des variantes de ces fonctions gourmandes en ressources processeur et les associer de manière dynamique au programme en cours, puis les profiler de manière itérative pour une efficacité comparative.
- *Régulateur de vitesse*: Relier dynamiquement la variation qui s'est avérée la plus adaptée à l'exécution en cours.
    - Surveiller périodiquement ses performances et revenir à la passe d'optimisation si des changements importants sont observés.



(a) Appel initial direct vers `Foo()`.



(b) Appel redirigé vers la fonction moniteur, qui distribue les variations injectées.



(c) Appel direct de la variante la plus adaptée.

Figure 1 – Progression de la passe d'optimisation.

La Figure 1 illustre la passe d'optimisation en décrivant l'exécution de la fonction `Foo`. Le flot de contrôle original (compilé statiquement) est présenté dans la Figure 1a, et le flot dynamiquement lié `Foo_monitor`. La fonction apparaît dans la Figure 1b. Après avoir choisi la meilleure variation, FITTCHOOSER contourne le moniteur en y reliant directement les sites d'appel, comme indiqué dans la Figure 1c, puis passe au régulateur de vitesse.

L'évaluation expérimentale de FITTCHOOSER sur d'importants benchmarks de l'industrie montre une accélération jusqu'à 19%, même avec un répertoire limité de transformations de programmes, ce qui suggère que des gains supplémentaires pourraient être possibles si des techniques d'optimisation plus sophistiquées sont incorporées dans FITTCHOOSER.

OFSPER: Spécialiseur de fonctions en ligne

La *spécialisation de fonction* (également appelée *Procédure de clonage*) est l'une des techniques d'optimisation utilisées pour réduire le temps d'exécution d'une fonction. L'idée est, au lieu d'appeler une fonction générique pour tous les sites d'appels, d'appeler différentes versions de celle-ci en fonction des valeurs prises par les paramètres. Les sites d'appels d'une fonction sont divisés en groupes en fonction des valeurs prises par les paramètres et une version spécialisée de la fonction est produite pour chaque groupe. Chaque version est spécialement optimisée pour une catégorie d'arguments particulière afin de permettre son exécution plus rapidement que celle de la version générique d'origine.

Pour appliquer la spécialisation de fonction, connaître la valeur du ou des paramètres est la clé. Dans la plupart des cas, les appels de fonction ne contiennent pas de constantes en tant qu'arguments, ils ont plutôt des variables, comme dans `Foo(a, x)`. Dans de tels cas, il n'est pas simple de faire la spécialisation des fonctions car les valeurs des variables peuvent être inconnues. Avec une optimisation guidée par profiling, ayant une exécution simulée du code pendant la compilation pour connaître le comportement du programme, la valeur ou une propriété des paramètres peut être prédite. Toutefois, il se peut que cette solution ne soit pas toujours réalisable car le comportement prédit peut varier au moment de l'exécution. Il est donc très difficile d'appliquer la spécialisation de fonction pendant la phase de compilation statique, ce qui signifie essentiellement que la spécialisation serait plus efficace lorsqu'elle est appliquée dynamiquement en connaissant les valeurs exactes des variables.

OFSPER est un outil permettant la spécialisation dynamique de fonctions lors de l'exécution d'applications. La spécialisation dynamique de fonctions est une technique d'optimisation de programme dans laquelle la spécialisation de fonctions est appliquée à une application en cours d'exécution pour améliorer son temps d'exécution. Dans cette technique, les différentes versions de la fonction sont créées dynamiquement en fonction des valeurs réelles prises par ses paramètres. Comme il est très important de connaître la valeur réelle des arguments pour effectuer une spécialisation de fonction, elle sera plus efficace si elle est appliquée à un programme en cours d'exécution. En outre, une version optimisée plus spécifique au matériel peut être produite dans cette technique grâce à la connaissance de la plate-forme matérielle en cours d'exécution, par rapport

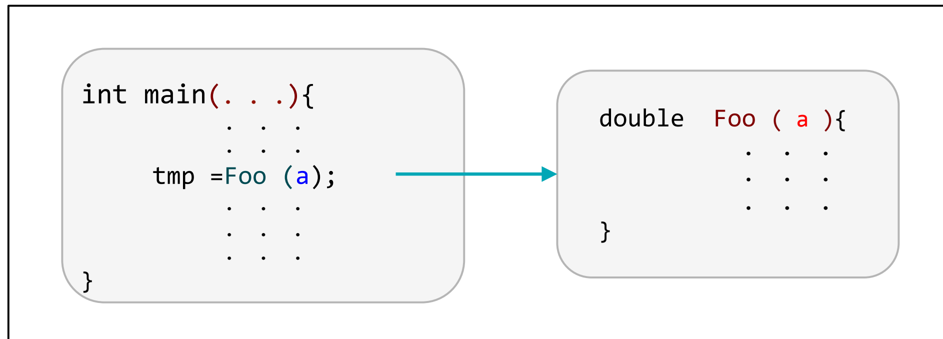
à la technique de spécialisation de fonction statique.

Une fonction peut être appelée depuis différentes parties du programme. Les valeurs prises par les paramètres pouvant différer d'un appel à l'autre, même du même site, il n'est pas possible de remplacer directement l'appel de fonction d'origine par un appel à une version spécialisée spécifique. Au lieu de cela, différentes versions doivent être maintenues en fonction des arguments. OFSPER utilise une fonction supplémentaire, appelée *monitor function*, pour gérer ces versions spécialisées et rediriger les appels de fonction vers les versions appropriées. Les *monitor function*, une pour chaque fonction, sont créées dynamiquement et nous remplaçons tous les appels de fonction d'origine par un appel à la *monitor function*. OFSPER est mis en œuvre sous forme de progression en quatre phases.

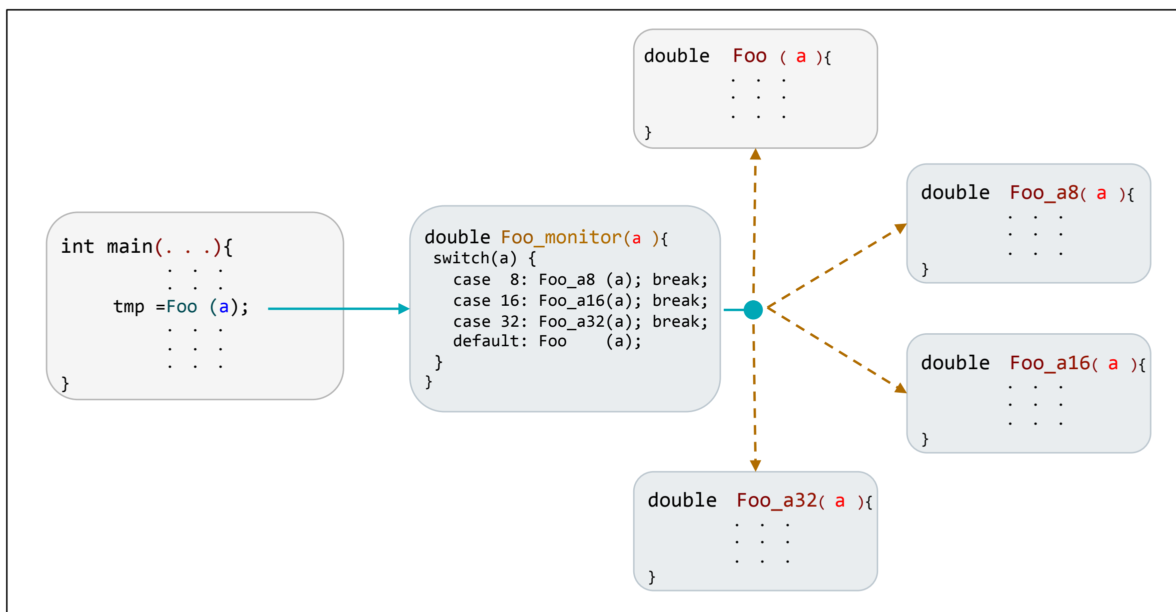
— *Profil*: Identifier les fonctions les plus gourmandes en ressources processeur au sein de l'exécution en cours.
— *Analyser*: Toutes les fonctions gourmandes en ressources ne sont pas spécialisables. Cette phase analyse les fonctions 'chaudes' sélectionnées et choisit celles qui conviennent à la spécialisation.
— *Surveiller*: Recueillir des valeurs prises par les paramètres de la fonction appropriée pour identifier la répétition des arguments.
— *Spécialiser*: Créer des versions spécialisées de la fonction pour la répétition d'arguments et les relier dynamiquement à l'application en cours d'exécution.

La Figure 2 montre la différence d'exécution entre les fonctions avant et après application de la spécialisation. Le flux de contrôle original (compilé statiquement) est présenté dans la Figure 2a, et la fonction liée dynamiquement `Foo_monitor`. La fonction apparaît dans la Figure 2b. `Foo_monitor` contrôle l'exécution des versions en fonction de la valeur courante du paramètre.

L'évaluation expérimentale de OFSPER sur d'importants benchmarks de l'industrie montre une accélération jusqu'à 35%.



(a) Exécution normale.



(b) Spécialisation de fonction dynamique

Tous les appels à function `Foo` sont redirigés vers `Foo_monitor` et `Foo_monitor` décide quelle version exécuter

Figure 2 – Séquence d’appel: Normal vs Spécialisation

Introduction

Problem Definition

A compiler is a software which converts human readable high level programs to machine readable low level programs. A simple compiler transforms each statement in the source code of the program to its corresponding binary code. However, modern compilers do transformations while converting the source code to binary such that the resulting binary code has better performance compared to the statement-to-statement transformed one. The whole compilation process, goes through various phases such as lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation. In this thesis we focus on code optimization phase which is meant to improve the program execution by means of reducing its execution time, energy consumption and/or code size. This phase may remove some unusable/unnecessary statements of the program, rearrange the statements or replace some statements to improve the performance. This optimizing functionality of compilers makes programming easier because developers can focus on features and readability, and let performance for the compiler to deal with.

Due to its nature, a static compiler has a limited visibility when it comes to taking into account the dynamic environment or behaviour of an application. Unknowns include actual input data that impacts the values flowing through the program and hardware-specific details. Program inputs can drastically change the efficiency of statically compiled code, yet in many cases is subject to total uncertainty until the moment those inputs arrive during program execution. Normally a static compiler partially applies data dependent optimization techniques by taking some rough estimates of the run time data, sometimes with the help of a profiling session carried out during static compilation. For example, loop optimization techniques like restructuring of loops, elimination of induction variables, loop unrolling and loop invariant code motion– all are applied with the assumption that the loop will be executing for a minimum number of times.

All of these alongside other optimization techniques like vectorization, loop tiling, unreachable code elimination, etc., could benefit from knowing the actual run time data. Run time data also help to minimize the dynamic instruction count along critical paths, to optimize branches for the typical case, maximize cache locality etc. [Fis81; PH90; HC89; EAH97; Wha99].

Modern hardware features can boost the performance of an application, but software vendors are often limited to the lowest common denominator to maintain compatibility with the spectrum of processors used by their clients. Given more detailed information about the hardware features, a compiler can generate more efficient code. But even if the exact CPU model is known, manufacturer confidentiality policies leave substantial uncertainty about precise performance characteristics. In addition, the activity of other programs colocated in the same runtime environment can have a dramatic effect on application performance. For example, if a shared cache is being heavily used by other programs, memory access latencies may be orders of magnitude longer than those recorded during an isolated profiling session, and instruction scheduling based on such profiles may lose its anticipated advantages.

A solution for these problems is to compile the source code after getting all the information about the program which different optimization techniques may require. We know that such a solution is impractical. A close to optimal solution is dynamic optimization in which the optimization is applied during the execution of the program. The main advantage of dynamic optimization is that the optimizer has full knowledge about the run time environment including hardware specific details. The optimizer also gets the input data also and so in many cases it can more precisely predict the future behaviour of the program.

One of the main challenges in doing dynamic optimization is lack of enough information about the program. This can be overcome by either (1) including necessary information in the binary code during static compilation or (2) using an intermediate representation (IR) of the program. Dynamic optimization generally causes run time overhead due to collection and management of profile data, analyzing that data and performing optimization. This overhead should be compensated by the speed up gained by dynamic optimization in order for the effort to be beneficial. Unfortunately, it is very difficult to anticipate how much gain can be obtained by an optimization without actually performing it.

Motivation

Decreasing execution time of a program is always an interesting subject in the research community. Plenty of work in both software and hardware level have already been done in pursuance of achieving this goal. Computer architects have introduced a lot of changes in the hardware field like vector operations, hardware counters, different levels of caches, etc. Modern compilers are very advanced in applying different optimization techniques to the program with or without the help of these hardware features. Knowing the run time machine characteristics is very helpful for a compiler to produce an optimized code. The availability of a vast variety of hardware in the market makes it difficult for a static compiler to produce machine dependent codes suitable for all the systems. This implies that recompiling the program on the running machine can produce better machine dependent code.

Predicting the run time behaviour of a program also helps static compiler in doing optimizations. Normally, a profile-directed static compilation is used to do so. The prediction is based on profile information collected from a number of test runs conducted during the static compilation. The main issue in this case is the correctness of the predicted data. The inputs used in the test runs may not be a 'typical' input. Finding the 'typical' inputs is very difficult for some applications. So the data collected from these test runs can go wrong during the actual execution. However a dynamic optimizer does optimizations based on the actual input data and so the probability of getting negative impact is very low.

Function specialization can be applied when a function argument is found to take same value through out different calls to the function. A static compiler has limited visibility about the actual run time values taken by an argument to a function. This makes it more complicated for a static compiler to apply function specialization. There are some cases where the call site itself contains the argument value, like $foo(a,\ 8)$. In such cases a static compiler can create one specialized version, say foo_8, of function foo by substituting the value 8 to its second argument. And then the call $foo(a, 8)$ can be replaced with the call $foo_8(a)$. Meanwhile, a dynamic optimizer can monitor all the values taken by the arguments and can act accordingly during the execution.

Programs in binary are not able to automatically adjust with the changes in the hardware. There exist many executable programs which all are 'old' but still usable and have

originally been compiled for now outdated processor chips. These may benefit from the new features provided by the modern processor chips but they need to be recompiled to use this extra features of the new hardware. This recompilation may be costly and will be very difficult when the source code is not available anymore. However, it may be relatively better to recompile only the critical functions and replace the original function with more optimized one.

Proposed Solutions

Compilers can produce more optimized versions of the executable once we provide the target machine characteristics. Nowadays, a wide variety of computers are available in the market. They are different in their behaviour and architecture. Because of this no compiler can produce one executable file which gives the same performance on all machines. This means that we need to produce separate executable file for each target machine. Compiling the whole program on each target machine is a hectic job. Instead, we propose a light weight solution where our tools recompile only the most critical functions. The term critical refers to the functions which take more running time.

This thesis proposes two independent tools, FITTCHOOSER and OFSPER, which apply dynamic optimization at function level. The base idea is to replace an existing function with an optimized version of the same. FITTCHOOSER, a dynamic feedback based fittest optimization chooser, helps to find a better optimized version of a function by comparing the performance of different versions of the function created dynamically. OFSPER, an online function specializer, dynamically specializes a function according to its live argument values. A small outline of both tools follows.

FITTCHOOSER

Most of the optimization techniques are not guaranteed to give good performance on every run time environments. For example, the loop optimization technique ‘loop unrolling’ might be more effective with unroll factor 4 in one case, where as in some other case unroll factor might need to be 2 for better performance. FITTCHOOSER tries to figure out the most suitable optimization for the current execution environment by considering different possible optimization techniques. For each critical function, it dynamically creates differently optimized versions of the function and monitors their per-

formance to get the fittest one. This fittest version will be used for the subsequent calls to the function.

This work was presented in the *HPCS 2018 - 16th International Conference on High Performance Computing & Simulation - Special Session on Compiler Architecture, Design and Optimization* [Ap+18] and will be explained in detail in Chapter 2.

OFSPER

Creating specialized code of whole program or part of the program is one of the optimization techniques used by the compilers in order to reduce the execution time of the program. There are some cases, where the compiler knows that certain invariants are true for a particular case or a particular running environment such that it creates a specialized codes only for such cases/environments by considering the behaviours of those invariants. *Function specialization* is such a specialization technique in which different specialized versions of a function are created and included in the program. Here, the specializations are based on the values of the arguments to the function. The idea is that instead of calling the actual function from every call site, call different versions of it according to the values taken by the parameters. Since function specialization requires the value of the arguments, it makes more sense if it is applied during the execution of the program where all values are known.

This work was presented in *International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation* [AR17] and will be explained in detail in Chapter 3.

Organization of Thesis

The reminder of the thesis is organized as follows: Chapter 1 provides background information needed for better understanding this thesis work. A brief overview of the latest research works in function level dynamic optimization is also presented in this chapter. Chapter 2 presents FITTCHOOSER, a tool for finding the fittest optimized version suitable for the current running scenario of the functions in the application. This chapter describes the idea behind FITTCHOOSER and its implementation details. Chapter 3 presents OFSPER, a tool to dynamically specialize the functions in a running applica-

tion leveraging the knowledge of actual function parameters. A piece of more detailed information about the implementations of both FITTCHOOSER and OFSPER is given in Chapter 4. Finally, Chapter 5 concludes this thesis with possible future works.

CHAPTER 1

BACKGROUND

This chapter presents the required background on various code optimization techniques and tools used in this thesis. Section 1.1 outlines common traditional compiler optimizations, and Section 1.2 details some of the previous attempts to overcome the disadvantages of traditional static compilation. And finally, Section 1.3 explains a library called PADRONE, which helps FITTCHOOSER and OFSPER to dynamically analyze and optimize the application processes.

1.1 Traditional Optimization Techniques

Modern optimizing compilers not only transform human-readable high-level program to machine-readable low-level binary code but also apply optimizations to improve the program. Such optimizations are targetted at reducing execution time, energy consumption or code size. Plenty of such code transformation techniques are already present in many compilers (Aho et al [ASU86], Bacon et al [BGS94], Muchnick [Muc97], Cardoso et al [CCD17], etc.). Some of the most common techniques which may refer in the following chapters of this thesis are briefly introduced in this section.

Constant Propagation: Constant propagation is one of the most common and aggressively applied optimization technique. This technique propagates the constants through the program which helps compilers to do a significant amount of precomputation. Table 1.1 shows an example for constant propagation. The value of `x` is propagated to the third statement. While it may seem that the need for this optimization is due to sloppy programming, it is seldom so. Usage of various high level programming constructs such as '`#define`', '`constexpr`', 'non-type template parameters' can result

in expressions where the operands are constants. In addition, previous optimization passes may resolve the value of certain variables to be constants.

Actual Code	After Constant Propagation
`int x,y;`	`int x,y;`
`x = 10;`	`x = 10;`
`y = x + 15;`	`y = 10 + 15;`

Table 1.1 – Constant Propagation

Constant Folding: Constant folding replaces the expressions with constant values as operands with its result. This optimization goes hand-in-hand with the constant propagation optimization. For example, the expression `y = 10 + 15` in Table 1.1 will be replaced with `y = 25` during constant folding. Compilers should be careful while applying this optimization, especially when cross-compiling, as the validity of this optimization is target-dependent. For example, the rounding mode of the machine used to compile may not be identical to the machine used to run the program; hence the results may not be identical [Cli90; SW90]. Hence some compilers only performs this operations for integer expressions, while others rely on multiversioning (mentioned later).

Algebraic Simplification: This technique simplifies some arithmetic expressions in the program by applying algebraic rules to them. For example, the expressions $x * 1$, $x + 0$ and $x/1$ - all can be simplified to a single x.

Strength Reduction: This technique tries to reduce the strength of the expression by replacing an expensive operator with an equivalent but less expensive operator. For example, in the expression `x*2`, the multiplication operator can be replaced with an addition `(x+x)` operator or a shift operator, and the exponential operator in x^2 can be replaced with a multiplication operator `(x*x)`.

Unreachable-Code Elimination: Unreachable-code segments are code segments which will never be executed. These segments are often created as a result of various optimizations such as constant propagation. They usually come in two ways, through conditional statements and loops. In the case of conditional statements like `if else`, if the predicate is statically known to be true or false, then the compiler can remove the

code corresponding nonexecutable branch. Similarly, for a given loop, if the compiler can prove that the number of loop iterations is zero, the code corresponding to the entire loop body can be removed.

Listing 1 Example Loop

```
for (i = 1 ; i <= 100 ; i++ ){
    c[i] = a[i] + b[i];
}
```

Loop Vectorization: Many modern CPUs have 'vector' or SIMD hardware unit which simultaneously performs operations on a set of data. Vectorizing the code can help to exploit these vector units. Listing 2 shows an example of vectorized code. Compared to the original version, the vectorized code can perform 4 operations simultaneously. Typical vectorization units only support stride 1 (or stride 0) memory accesses. While modern vectorization units solve this problem using scatter/gather instructions, they are still not efficient as stride 1 (or stride 0) access.

Listing 2 Loop Vectorization

```
// original code
for (i = 0; i < 100; i++)
    c[i] = a[i] + b[i];

// vectorized version
__m128 rA, rB, rC;
for (int i = 0; i < 100 i+=4){
    rA = _mm_load_ps(&a[i]);
    rB = _mm_load_ps(&b[i]);
    rC = _mm_add_ps(rA,rB);
    _mm_store_ps(&C[i], rC);
}
```

Loop Fusion and Loop Fission: In loop fusion, loop bodies of two different loops are combined into a single loop. Loop fission is the opposite of loop fusion; it divides the loop body of a single loop into two loops. [KM94]. There are several advantages like increasing level of parallelism, data locality for loop fusion and like increasing the potential for loop pipelining and loop vectorization for Loop fission [CCD17]. Since these two transformations may change the order of execution of operations, they can be applied only when the preservation of data dependencies are ensured. Listing 3 shows

an example where the original code cannot be parallelized or vectorized due to data dependencies and how loop fission can help in parallelizing/vectorizing this code.

Listing 3 Loop Fission

```
 // Original code
 for (i = 0; i < N - 1; i++) { //neither parallelizable nor vectorizable; RAW S1->S2
     w[i+1] = x[i] + 1; S1
     y[i] = 2 * w[i];    S2
 }

 //After loop fission
 for (i = 0; i < N - 1; i++) { //parallelizable and vectorizable
     w[i+1] = x[i] + 1; S1
 }
 for (i = 0; i < N - 1; i++) { //parallelizable and vectorizable
     y[i] = 2 * w[i]; S2
 }
```

Loop Unrolling: Loop unrolling improves the performance of a program by reducing loop overhead, increasing instruction parallelism and improving register reuse, data cache or TLB locality [BGS94]. The idea behind this technique is to reduce the total number of iterations of the loop by repeating the body of the loop multiple times in a single iteration. The number of times the function body is replicated, known as $unrollingfactor$, effect the performance [SA05]. Listing 4 shows the unrolling of the `for` loop used in Listing 1 by a factor of 4. Compared to the original code, the number of compare operations in the unrolled is reduced by a factor 4.

Listing 4 Loop Unrolling

```
    // unrolled code
    for (i = 1 ; i <= 100 ; i += 4 ) {
        c[ i ] = a[ i ] + b[ i ];
        c[i+1] = a[i+1] + b[i+1];
        c[i+2] = a[i+2] + b[i+2];
        c[i+3] = a[i+3] + b[i+3];
    }
```

Unrolling can also be applied to outer loops. However, typical outer loop unrolling is followed by the jam (fusion) operation. As an example consider Listing 5 which shows the original, outer loop unrolled version, and the unroll + jam-ed version. The unrolled

Listing 5 Loop Unroll and Jam

```
// original code
for(i = 0; i < 100; i++)
    for(j = 0 ;j < 100; j++)
        y[i] = y[i] + a[j][i] * x[i];

// 4 way unrolling of i loop
for(i = 0; i < 100; i+=4)
    for(j = 0 ;j < 100; j++)
        y[i] = y[i] + a[j][i] * x[i];
    for(j = 0 ;j < 100; j++)
        y[i+1] = y[i+1] + a[j][i+1] * x[i+1];
    for(j = 0 ;j < 100; j++)
        y[i+2] = y[i+2] + a[j][i+2] * x[i+2];
    for(j = 0 ;j < 100; j++)
        y[i+3] = y[i+3] + a[j][i+3] * x[i+3];

//unroll and jam
for(i = 0; i < 100; i+=4)
    for(j = 0 ;j < 100; j++){
        y[i] = y[i] + a[j][i] * x[i];
        y[i+1] = y[i+1] + a[j][i+1] * x[i+1];
        y[i+2] = y[i+2] + a[j][i+2] * x[i+2];
        y[i+3] = y[i+3] + a[j][i+3] * x[i+3];
    }
```

version just benefits from reduced comparison operations. However, the unroll + jam-ed version is much more cache and vectorization friendly.

Loop Tiling: Loop tiling is a powerful technique to improve data locality and reduce the number of cache misses when the data footprint of the loop exceeds the cache capacity. Instead of operating on entire array, this technique divides the big array into chunks which are small enough to fit in the cache. Since the operations are carried out on these small blocks, the data used in the loop stays in the cache until further reuse. Not only the size but also the shape of these small blocks have a significant role in performance gain. They need to be decided based on the running environment [KKO00; Pén+16; HS02; GSK01].

Loop tiling is also beneficial when the data-layout does not align with the data access pattern. For example, when a row-major array is accessed along the columns. Matrix

Listing 6 Matrix Multiplication

```
for(i = 0; i < N; i++)
    for(j = 0; j < N; j++)
        for(k = 0; k < N; k++)
            ans[i][j] = ans[i][j] + matrixA[i][k] * matrixB[k][j];
```

multiplication is one of the good example for applying loop tiling. Listing 7 shows the tiled version of the matrix multiplication code given in Listing 6. The main disadvantage of loop tiling is the increased overhead due to extra loops. Assuming that the original code has N loops and there are M levels of fast memory (e.g., caches, scratchpad memory, registers), most generic tiling will result in $N \times M$ loops.

Listing 7 Loop Tiling on Matrix Multiplication

```
for(bigJ = 0; bigJ < N; bigJ += jBlockSize)
    for(bigK = 0; bigK < N; bigK += kBlockSize)
        for(i = 0; i < N; i++)
            for(j = bigJ; j < min(bigJ + jBlockSize, N); j++)
                for(k = bigK; k < min(bigK + kBlockSize, N); k++)
                    ans[i][j] = ans[i][j] + matrixA[i][k] * matrixB[k][j];
```

1.2 Prior Works

Traditional compiler optimizations are designed for generic codes. Hence, the compile-time and run-time overheads of these optimizations are typically very low, and most of these optimizations are rightfully very conservative. However, many advanced optimizations, such as Multi-versioning and Profile Guided Optimizations (PGO), are much more aggressive and may have slightly higher overheads. Such code optimization techniques can be broadly divided into (i) Static optimizations and (ii) Dynamic optimizations. Static optimizations, which rely solely on the compile-time information, are applied during the compilation or link phase of a program. Traditional compilers almost exclusively rely on static optimizations to optimize the code. Dynamic optimizations are on-the-fly optimizations that rely on run-time information, such as memory address accessed. It could also take advantage of static information.

1.2.1 Static Optimizations

Traditional static optimizer's sole reliance on compile-time information along with conservative assumptions limits optimization opportunities and effectiveness. Since the exact micro-architecture may only be known at runtime, typical codes, including several heavily used libraries, are compiled for generic architectures whose Instruction Set Architecture (ISA) is supported by all the target platforms. While this approach achieves good generality, it may heavily hamper several optimizations, such as vectorization. Assume that the generic architecture does not support vectorization. In the case, even though the actual target may support vectorization, the produced binary cannot take advantage of vector units. Several existing works described below try to alleviate these fallbacks.

Multi-versioning

Function multi-versioning is one way of handling the difficulty in producing executable suitable for different architectures [CDS03]. In this technique, frequently executed functions are compiled to different versions for different kinds of architectures. For that the developer tells the compiler to create different versions of a function by giving the architecture name as attributes. For example, the attribute

```
__attribute__((target_clones ("avx2", "arch = atom", "default")))
```

tells the `gcc` compiler to create three versions of the function; one for AVX2, one for Intel atom and one common version for all other architectures.

One of the main disadvantage of this technique is that it requires the help from the developer to create different versions. The developer should be aware of the different features, which may be useful for his/her function, provided by the architectures to determine the number of versions need to be created. This technique fails to produce new versions in the future based on the introduction of new architectures without modifying the source code.

Multi-versioning can also be applied with the help of Feedback-Driven program Optimization (FDO). FDO tries to provide some dynamic information along with the static information to help the compiler to make better optimization decisions by collecting the behaviour of some training runs of the program [Che+10; Smi00]. The multi-versioning

uses FDO to create versions of the function based on the profile collected on test runs on representative inputs. The compiler then includes special instructions into the executable for selecting the appropriate version during the run time. One of the main disadvantage of the FDO based compilation is its build time. It may require a normal initial compilation, then test runs on different inputs and finally a final compilation to get the executable containing different versions of the function. As an example consider loop parallelization. Representative inputs can be used to determine the minimum loop trip count for loop parallelization to be beneficial. The compiler can then create a parallel version and sequential version of the same loop, and a switching mechanism to select the appropriate version based on the above-said loop trip count. The main challenge in this technique is finding the appropriate execution environment and input sets. For a big transaction processing application, it may require to setup a database and representative set of queries for the test run. Creating such an environment can be very difficult. Even though the application executed on a number of different data sets during profiling, there is still a chance that none of these data sets may not actually reflect the application's real usage.

The other side effect of multi-versioning is code size increase of the executable. When the versioning applied on most of the functions, the code size will increase linearly. This can be reduced by selective multi-versioning in which the versioning is applied only on selected functions. Zhou et al [Zho+14] present a couple of solutions for applying selective multi-versioning with the help of FDO. Ekemark [Eke16] explained how multi-versioning can help to reduce the power consumption.

Iterative Compilation

This technique tries to find a best optimization technique for a program by compiling the program repetitively. After each compilation, the program is executed to analyze the effect of compilation. Figure 1.1 taken from [KKO02] depicts the Iterative Compilation process. Based on the feedback from the previous run, the transformations for the next compilation are applied. Due to this repetitive nature, the compilation time dramatically increases. Even though this technique succeeds in finding a good optimization for the given machine, we cannot rely on this to produce executables for a variety of architectures.

Kisuki et al [Kis+00; KKO02] show the effectiveness of iterative compilation for se-

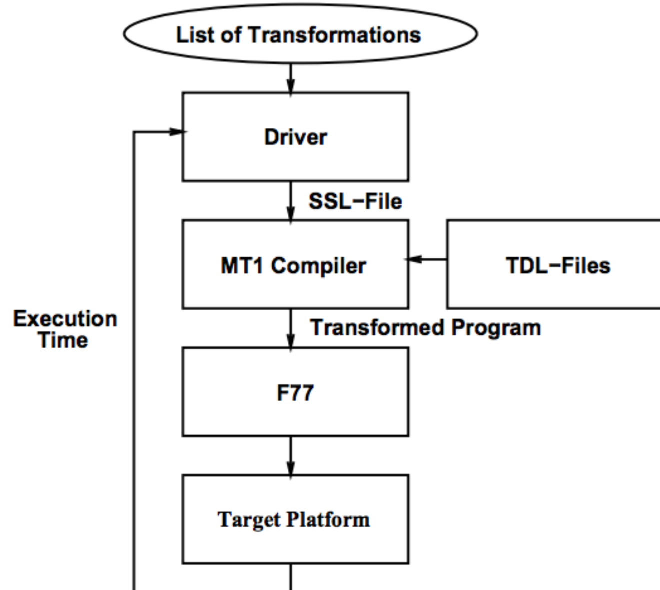


Figure 1.1 – Iterative Compilation Flow Chart given in [KKO02].

lecting best Loop Tilling size, Loop Unrolling factor and Loop padding size. In their experiments they required around 350-400 iterations to find a satisfactory optimization. They say this method will be very useful for embedded applications where the compute intensive kernels themselves are small and fast. The time taken for the compilation is the main disadvantage here. And this technique also rely on the sample inputs used in the iteration. They may not perfectly reflect the actual run time data. The selected optimization matrices may have negative effect on the actual data.

Memoization

Memoization is an optimization technique in which the result of a program section is stored and reused in the future when the same execution sequence repeats. This can be applied at instruction level, block level, region level or at function level. For example, consider the mathematical function `sin`. The return value of this function purely depends on the input argument. If the input is same across different invocations of the function, then there is potential to improve the application execution time by saving the result of one invocation and reusing it for the later ones there by avoiding the requirement of repeated executions.

Suresh et al [Sur+15] discuss about function memoization in software for dynamically linked functions. The idea is saving the result of a function in a table indexed by XOR hash function of its arguments and return these results when the argument repeats. Memoization is implemented by intercepting the dynamically linked function calls using LD_PRELOAD technique. An extension of this work [SRS17] extends this function memoization to user defined functions as well as by implementing the function memoization at compile time using the LLVM framework. These works can be treated as an extreme case of *function specialization* which we will discuss in Chapter 3.

Discussion

Since static optimizers are invoked during compile/link time, the application runtime is not affected by the optimization overheads. Thus, when compared to dynamic optimizers, static optimizers can use expensive analysis and code generation passes to optimize the code. However, the applicability and effectiveness of static optimizations are restricted by the conservative assumptions made by the compiler and lack of information such as the exact target architecture, hardware parameters (e.g., cache sizes, the run-time behavior of the program), etc.

1.2.2 Dynamic Optimizations

Dynamic optimizations are optimizations that are applied during the application runtime. In contrast to static optimizers, dynamic optimizers can take advantage of runtime information such as the exact execution environment (e.g., ISA, cache size) and program behavior (e.g., memory access pattern, value of function parameters). Thus, dynamic optimizers can even optimize codes to efficiently execute on architectures, which may not even be available when the binary was generated. The ever-growing importance of cloud-computing platforms, where the precise target architecture is only known during application deployment, highlights the importance of such optimizations. Mobile apps, which should run on possibly hundreds of different architecture, represents another important and practical use case for dynamic optimizations. In addition, techniques such as Just-In-Time (JIT) compilation, can apply modern optimization passes on legacy codes which might have been compiled with an old compiler that does not support many optimizations or might have missed many optimizations.

Dynamic Recompilation

The main advantage of dynamic optimization is the knowledge of the underlying hardware. Powerful optimizations like vectorization require hardware support. Compiling programs specific to all available machines in the market is impractical. However, recompiling small sections of the programs specific to the hardware makes it more practical. This can be achieved by shipping an intermediate code of the program produced by the compiler along with the binary. And this intermediate representation can be used to recompile the critical functions of the program during the execution.

Nuzman et al [Nuz+13] discusses about dynamic function level optimization with recompilation. They use both native executable file and intermediate representation (IR) file of the program. During execution, they recompile hot methods from IR file using a Java JIT compiler and the recompiled versions are stored in a *Code Cache*. With the help of a trampoline created at the beginning of the original function body, the function calls are redirected to the new recompiled version.

Sukumaran-Rajam et al. [Suk+14; SC15] describe a Polyhedral model based dynamic optimization framework called APOLLO. During the compilation phase, APOLLO creates an instrumented code version along with several optimization templates. The instrumented version is used to collect profiling data such as memory address accessed and loop bounds. The optimization templates are code structures that support several optimizations such as parallelization, loop tiling, and loop interchange, etc. However, the exact code optimization is only determined during the application run-time. When the actual code is executed, the program behavior is captured by using the instrumented version on a small number of iterations. The dependencies are then computed using the profiling data, and the code transformations are speculatively computed by invoking Pluto – a Polyhedral optimizer at run-time. To handle misspeculation, special memory backup mechanisms are employed to restore the program state to a safe state. The framework is capable of performing optimizations such as parallelization, loop tiling, loop permutation, loop skewing, etc.

Profile Guided optimization

Modern static compilers can apply a large number of optimization techniques to the program. Applying a typical optimization consists of mainly two tasks: uncovering the

optimization opportunities; and applying them. The main limitation of static compiler in applying some of the optimization effectively is the unaware of program's run time behavior. In profile guided optimization, the compiler collects profiling data by carrying a certain number of test runs during the compilation. Using these data, compiler predicts the common behavior of the program like the most executed sections of the program, and tries to apply optimization techniques suitable for them. As an example consider branch prediction and branch code placement. The compiler can use the profiling data to predict the branch-taken probability which can be used to rearrange the code to increase hardware branch predictor accuracy, reduce the number of taken 'jump' instructions or to reduce the likelihood of instruction cache misses.

Discussion

Dynamic optimizers take advantage of rich run-time information such as exact target architecture and program behavior to optimizer codes. Unlike static optimizers, since dynamic optimizations are applied at run-time, the optimization benefits should outweigh the optimization cost. In other words, dynamic optimizations should be fast and lightweight. It should use less processing power and memory as both these resources are shared with the application. For single-threaded applications, an extra core can be used for dynamic optimizations. This approach will help to hide the overhead of dynamic optimizations. However, this option is not available for multi-threaded code as the cores are possibly used to run the application itself. In addition, for multi-threaded applications, all threads may have to be synchronized before modifying the code.

1.2.3 Proposed work

The proposed work operates on the binary without any special information being transferred from the static compilation phase. This work tries to propagate the possibilities of static compilation to the execution time of the program. Rather than introducing one new optimization technique, it mainly concentrates on introducing a different approach in applying the existing ones. Moreover, unlike other dynamic optimizers, our tools try to protect the program transparency[BZA12]. By touching the program as little as possible, they allow the applications to run their own. With the help of PADRONE, our tools attach to the application process just for a small amount of time to make the required changes and detach after that to allow the process to run its own. If any requirement

arises in the future, they can attach and detach again.

Challenges

While optimizing binary codes offers many advantages they are very challenging. Unlike source/intermediate code optimizers, binary optimizers don't have access to the rich semantic application information. Most of the semantic information is lost during the binary code generation phase. Basic code structures such as loops, Control Flow Graph (CFG), etc. have to be rebuild from scratch. Even disassembling binary codes for complicated ISA, such as x86, is challenging due to unaligned and variable length instructions. With recent emphasis on security, several hardware/software security mechanisms may make dynamic code generation and code patching very difficult.

1.3 PADRONE

PADRONE is a library created by Riou et al [Rio+14] which provides APIs for dynamic binary analysis and optimization. It can attach to a running process to carry out different activities thanks to the `ptrace` system call, in a way similar to *gdb*, the GNU Project Debugger. PADRONE has functionality for doing profiling on the process for well understanding of the program's behavior. The profiling can be used to recognize the hot spots of a process in which it spends most of its CPU time. The profiling collects samples by probing the program counter in regular intervals. A user can set the duration and frequency, number of samples per seconds, of a profiling session. Each sample contains an instruction pointer which is used to retrieve the address of the function. The functions with most number of samples are considered as critical functions.

One of the main functionality of PADRONE is its ability to include new functions to the process by creating a code cache [1] in the process memory space. The new function can be injected as normal binary or as a shared library. In normal binary case, we can directly write the binary code of the function in the code cache by calling the function `padrone_cc_insert` provided by PADRONE's API. When we insert as a shared library, PADRONE writes a small code into the code cache which calls `dlopen` to insert

1. A code cache is a contiguous memory area in the application's memory heap. PADRONE creates it by writing a small piece of code into the application's code segment which calls `malloc` to allocate the required memory.

our function as shared library into the application process. After successful injection, PADRONE can redirect a function call to this newly injected function. To achieve this, PADRONE replaces the first instruction of the original function with a `trap` instruction. At the next invocation of this function, the process receives a signal and stops. PADRONE can then fetch the return address to find the address of the call site and can replace it with a call to the new function. Future executions of the call site will reach the function placed in the code cache. By using this functionality of PADRONE we can replace a function with a more optimized one.

The other useful feature provided by PADRONE is its monitoring functionality. This functionality is used to monitor a function in the process for a period of time. For any function *foo* in the process we can inject a new function *foo_monitor* to the process as a shared library. The *foo_monitor* executes on every call to the function *foo* during a user-defined time period. After that the process executes normally as before. The *foo_monitor* will be able to call the original function *foo*. PADRONE redirects all calls to *foo* to *foo_monitor* except the ones coming from *foo_monitor* itself.

PADRONE has several advantages compared to other binary translators. Instead of whole program, it allows you to concentrate only on chosen functions. It injects optimized function as a contiguous block of code, whereas a binary translator will often fragment a function, reducing code locality. Binary translators typically replace the return instruction with a push/jmp sequence which interferes with the branch predictor. The internal implementation of PADRONE does not require intercepting any system calls or standard library calls, although it is always able to do so on demand. Moreover PADRONE runs in a separate process, allowing it to execute its own analysis on a different core (when available).

CHAPTER 2

FITTCHOOSER: A DYNAMIC FEEDBACK-BASED FITTEST OPTIMIZATION CHOOSER

2.1 Introduction

The introduction of advanced features like hardware counters and vector processing units in modern microprocessors can enable programs to run faster without requiring changes to the source code. For example, a vector processing unit can operate on an entire array of data in a single instruction. Programs can even be optimized at runtime by dynamically monitoring hardware counters [Leh16; Wic+14; SG06a; Wat+17]. But software vendors must always take hardware compatibility into consideration before enabling these advanced features in their applications. Many processor models do not support all the latest optimization features and will raise a hardware fault if a program attempts to invoke them. But even if the vendor could compile the program directly on each deployment machine separately—which is highly impractical—today's best commercial and open-source compilers often miss optimization opportunities. This is due in part to lack of public information about the low-level details of CPU features. Without a precise model of the processor's performance characteristics, the compiler resorts to heuristics for selecting such essential factors as the number of loop unrollings or the scheduling of load instructions [Mac+17]. Our experimental results in Section 2.4 show that these heuristical models do not always make the best choice for a given program and hardware environment.

We have developed a dynamic optimization tool called FITTCHOOSER to overcome

these limitations by generating variations of the program's machine code and empirically evaluating the variations to select the best performer. This iterative process allows FITTCHOOSER to find the most suitable optimization technique for a program's most processor-intensive functions in its current runtime environment. To account for potential changes in performance characteristics, which could for example be caused by expansion of a frequently traversed array beyond the capacity of the L3 cache, FITTCHOOSER continuously monitors its optimized functions and restarts the evaluation process when significant changes are observed.

Although our experiments show that FITTCHOOSER is efficient enough to recover its own overhead where it discovers effective optimizations for the target program, it may not always be practical to run the program under FITTCHOOSER. For example, SPMD programs run in parallel on all cores of a machine, while FITTCHOOSER anticipates that it can run on a separate core to mask the majority of its overhead. In such cases the user can conduct a preliminary tuning phase where FITTCHOOSER discovers the best optimizations for the machine, and then deploy those optimizations using our extension of the Linux loader called the FITTLAUNCHER. Although this approach does not benefit from the per-execution tuning of FITTCHOOSER, it does integrate the selected optimizations without the overhead of profiling and monitoring.

The current version of FITTCHOOSER explores a limited set of optimizations based on ordinary features of popular compilers, often making better use of those features than the compiler itself. Future enhancements to FITTCHOOSER could expand its repertoire to include advanced optimizations reported only in research, along with new experimental optimizations developed specifically for the tool. The remainder of this chapter focuses on the FITTCHOOSER infrastructure and presents the currently available optimizations as a proof of concept that in our experience works in practice. Section 2.2 begins with introducing the idea behind FITTCHOOSER and Section 2.3 describes the implementation. We report the results of our experiments in Section 2.4, which include the overhead of FITTCHOOSER along with key examples of successful optimizations. Section 2.5 presents related work and Section 2.6 concludes.

2.2 Survival of the fittest

The performance of a given execution of a program can be affected by a broad range of factors, making it difficult to determine in advance which optimization techniques may be the most advantageous. Many of these factors can be entirely unpredictable, for example if the program processes an input stream representing end-user activity, it may not be possible to predetermine the ideal optimizations for a given period of that input stream. Even if a compiler were to choose the ideal optimizations for a given execution scenario, the same compiled program could be executed in a slightly different scenario where other optimizations would improve performance. To bridge this gap between compile-time optimization and a concrete program execution, FITTCHOOSER employs dynamic instrumentation to generate and test various combinations of optimizations at the beginning of program execution and then transform the program to use the combination that empirically proves itself to be the most effective. This is implemented as a progression through three phases:

— *Profiling*: Identify the 5 most processor-intensive functions within the current execution.

— *Optimization Pass*: Generate variations of those functions and dynamically link them into the running program, then iteratively profile each one for comparative effectiveness.

— *Cruise Control*: Dynamically link the variation that proved to be the most fit for the current execution.

  — Periodically monitor its performance and return to the Optimization Pass if significant changes are observed.

2.2.1 Profiling

The key advantage of FITTCHOOSER over optimizating at compile-time is that it can precisely discover the program's performance characteristics, not just for a given machine, but also for a specific execution. This comes at the cost of runtime overhead to modify the machine code while the program is performing its tasks. To avoid squandering potential speedups, FITTCHOOSER profiles the application to identify the five

Listing 8 Example Function

```
double Foo(unsigned int a[], unsigned int b)
   for i = 1 to  b
       c += a[i]/b;
 return c;

 int main (int argc, char *argv[] )
   - - -
  for i = 1 to  10000
      - - -
      temp = Foo(a,i);
      - - -
  - - -
```

functions in which the processor spends the majority of its time. These few *critical functions* are selected as exclusive candidates for optimization.

2.2.2 Optimization Pass

This phase begins with an analysis of each critical function to determine which program transformations can potentially be advantageous. For example, functions containing loops are typically candidates for loop unrolling and loop tiling. Conversely, functions containing `static` variables are not eligible for these optimizations because of the difficulty in preserving the value of the variable across different variations of the function. The optimization repertoire of FITTCHOOSER is presented in detail in Section 2.3.

For each candidate optimization, a new version of the selected function is generated and injected into the running process. To compare the performance of the variations, we inject a meta-function `monitor` that acts both as a dispatcher and a timer. The monitor rotates between the injected variations in round robin fashion to maintain timing fairness. Each variation is allocated a fixed (configurable) number of invocations per round, and evaluation continues until the total number of invocations reaches a fixed (configurable) threshold. At the end of this evaluation period, the monitor functions are retired by patching calls directly to the best-performing variation.

The pseudocode in Listing 8 illustrates a common scenario where a single progression through the variations would result in unfair evaluation. Since the number of iterations of the `for` loop in function `Foo` depends on parameter `b`, the value of the parameter `b`

affects the running time of the function Foo. The first 100 calls to the function have an average of 50 iterations, while the second 100 calls have an average of 150 iterations. Suppose we run the first version for the first 100 calls and the second version for next 100, then comparing the average running time of them to find the fastest is unfair. Instead, executing them in a round robin fashion with a quanta of 10 calls makes it more comparable.

While there are other ways to maintain fairness of evaluation, the monitor function has been implemented with a round robin strategy to avoid complications with low-level timing measurement. An alternative approach could measure flops, but this generally requires hardware counters that may not be available on older processor models. Another option would be to measure instructions per second, but this will be inaccurate for optimizations that reduce the number of executed instructions (for example, loop unrolling may eliminate a significant number of branch instructions).

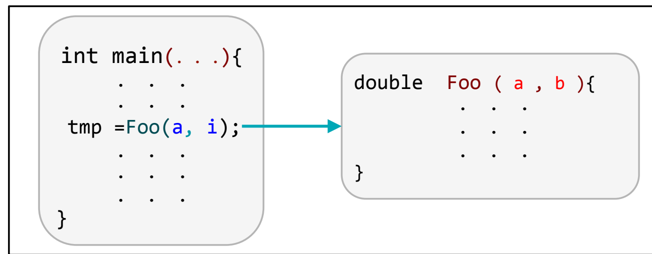
Figure 2.1 illustrates the Optimization Pass by depicting the execution of function Foo when the example program given in Listing 8 is executed under FITTCHOOSER. The original (statically compiled) control flow is shown in Figure 2.1a, and the dynamically linked Foo_monitor function appears in Figure 2.1b. After choosing the best variation, FITTCHOOSER bypasses the monitor by linking call sites directly to it, as shown in Figure 2.1c, and then goes on Cruise Control.

2.2.3 Cruise Control

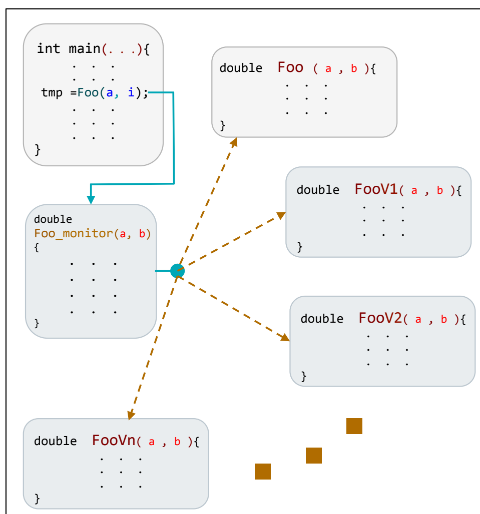
Application behavior may change during execution such that our selected program transformations may no longer be optimal. To maintain performance through these changes, FITTCHOOSER remains on Cruise Control throughout the execution of the program, periodically evaluating the optimized functions. If significant changes are observed, FITTCHOOSER revisits the Optimization Pass in search of the best variations for the present conditions.

2.3 FITTCHOOSER

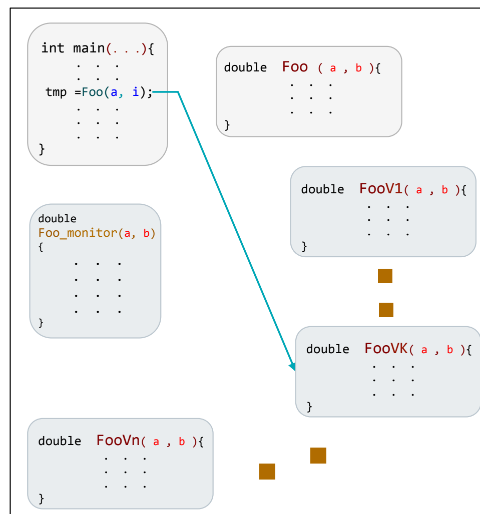
The three-phase strategy for finding and linking the fittest optimizations is coordinated by the FITTCHOOSER application, which runs in its own separate process. Inter-



(a) Original call direct to `Foo()`.



(b) Call redirected to the monitor function, which dispatches to the injected variations.



(c) Call direct to the fittest variation.

Figure 2.1 – Progression of the Optimization Pass.

process communication is facilitated by the binary instrumentation framework Padrone [Rio+14], which supports selective instrumentation of a target process while minimizing interference with its native flow of execution. Section 2.3.1 presents Padrone in more detail and makes a case for its fitness as the foundation of FITTCHOOSER. Section 2.3.2 moves on to the implementation of FITTCHOOSER, and Section 2.3.3 describes the lightweight deployment alternative FITTLAUNCHER.

2.3.1 Padrone

Since our goal is to improve performance while monitoring and modifying the program, it is essential for FITTCHOOSER to minimize its own overhead. While there are many tools that can facilitate the instrumentation, Padrone proves to be the least intrusive and therefore the most advantageous for conserving speedups. Where a typical binary translator takes full control of a program and translates every executed instruction, Padrone provides comparable instrumentation on a selective basis, affecting only the specific set of functions that are modified. An alternative approach would compile the instrumentation directly into the target application, but this can change critical performance factors such as code layout, and introduces the challenge of integrating with the build system of every target application. A self-contained tool like Padrone is more practical, communicating with the target binary over the the Linux `ptrace` API like an interactive debugger.

Padrone instruments the target process by injecting code changes, which involves modifying existing program instructions and/or generating code to a dynamically allocated code cache. It is also possible to inject a shared library via `dl_open()`. To link a new function into the running program, Padrone inserts a `trap` at the start of the original function to identify incoming calls (by checking the return address at the trap). After modifying the operand of the incoming call instruction, subsequent invocations of that call site will go directly to the new target function.

The advantages of Padrone over conventional binary translation are not limited to its selective instrumentation API and its isolation in a parallel process:

— Binary translators have a baseline overhead of at least 12% on the SPEC CPU 2006 benchmark suite [Haw+15], increasing to 30% or more for desktop applications [SZW06], and inflating up to 17× where a JIT engine is involved [HDT16].

In contrast, the baseline overhead of Padrone is negligible, consisting of just one interruption for `ptrace` attach.

— Padrone does not require global monitoring or modification of the effects of system calls or standard library calls, even if those effects are visible to the program.
— Binary translators typically replace the `ret` instruction with a `push/jmp` that interferes with hardware optimizations for call/return symmetry.
— Padrone's selective instrumentation allows greater control over the alignment and colocation of injected code by reducing pressure to consolidate the code cache.
— Many important hardware performance counters are local to a CPU core, allowing for accurate measurements even while other cores are highly active. This advantage is lost under in-process binary translation where the activity of the translator occurs on the same core as its target application and pollutes the counters. Padrone, on the other hand, is able to monitor performance counters in the target process via the PAPI function `PAPI_attach()` while limiting its own footprint on the counters to the relatively lightweight `ptrace` calls.

While it is possible to compile FITTCHOOSER directly into the target application, this is highly impractical for most deployed software, and impossible for legacy binaries that were compiled before Padrone was available.

2.3.2 Implementation

We implement FITTCHOOSER in plain `C` using Padrone's API for introspection and instrumentation of the running process. To optimize a program with FITTCHOOSER, the user first launches the program and then passes the process ID to FITTCHOOSER which connects via `ptrace` and begins the Initial Profiling phase. In its current stage of development, our FITTCHOOSER prototype also requires the LLVM IR [LA04] of the target program to be provided by the user. This inconvenience will be replaced by a technique to lift the program's machine code to LLVM IR. Previous work by Hallou et al. [HRC16] showed that the LLVM IR produced by using McSema infrastructure [DR] can be used for optimizations as complex as vectorization. A similar approach would leverage the decoder of the binary translator HQEMU [Hon+12], which presents as its fundamental contribution the transformation of the internal QEMU IR to LLVM IR such that the LLVM compiler can be used to optimize translated code on the fly. Given this future enhancement, FITTCHOOSER would no longer require the user to provide any

information about the target application.

Candidate Optimizations

The efficiency of FITTCHOOSER and its underlying framework Padrone are essential for realizing performance gains from an optimization that is constructed entirely at runtime. But the pivotal component of FITTCHOOSER is its code generator that produces the optimization candidates. Given an extensive repertoire of powerful optimizations, the potential speedup depends mainly on the selection of candidates that have a high probability of (a) significantly increasing performance of the target function while (b) maintaining near-native performance even in an unexpected worst-case scenario. While our current experimental results operate on a limited optimization vocabulary that focuses on standard loop unrolling, the integration of the LLVM compiler provides FITTCHOOSER with easy access to a broad range of optimizations available from the LLVM community. As more advanced techniques are incorporated into FITTCHOOSER, its analysis of the target function will need to be increasingly effective in identifying the most promising avenues of optimization while recognizing potential pitfalls that could incur unacceptable overheads during evaluation.

The analysis of the target function can potentially be complemented by dynamic profiling of performance counters (where available). For example, if a group of optimizations aims to reduce the frequency of a certain hardware operation, a dynamic profile of corresponding hardware counters could enable FITTCHOOSER to accurately estimate the potential of those optimizations for the current execution. Research has explored the use of hardware counters in profile-guided optimization [Che+10], including dynamic compilers such as JIT engines [SG06b], but these efforts report significant difficulty in correlating hardware events with specific program code fragments. These problems do not occur for FITTCHOOSER because, instead of speculating about the significance of hardware event counts, it can explore a hypothesis about potential optimizations by simply generating an exploratory variation of the target function and empirically observing the change (or lack thereof) in hardware events.

Variation	Count	Timings
1	2103	3210
2	2100	3021
3	2100	3310
Original	2100	3250

Table 2.1 – Example of the shared Monitoring table.

Monitor Functions

While the round robin dispatch of the monitor functions is relatively straightforward, two interesting challenges arise where the functions are integrated into the target program. The first is to compare the performance of the injected variations without encumbering the target program. The monitor function could easily perform the comparison directly, but this can involve a significant amount of computation, especially when there is analysis involved in determining what action to take next. Instead, the monitor function records the average execution time of each round to a table that is shared between the target program and FITTCHOOSER. An example of its contents is depicted in Table 2.1. The table is hosted in a System V shared memory segment, and shared between the two processes as shown in Figure 2.2. In our current implementation, synchronization is not required for table access because FITTCHOOSER waits until the end of the Optimization Pass and reads the entire table after the monitor functions have been removed. In the case that future enhancements involve intermediate evaluation of the variations, for example to adjust them during the Optimization Pass, it may become necessary to introduce a locking scheme.

The second challenge involves the pass-through of the return value from the target function to its caller within the target program. Ideally the intervening monitor function would simply make a tail call to the monitored function, allowing the return value to naturally pass through. But this is not possible because the monitor must stop its internal timer, and at the end of a round it must also update the shared table with the observed average execution time. Instead, the monitor function stores the return value from the target function, performs the necessary computations and updates, and then returns the value. This is difficult to do in a generic manner because the return value may take the form of a struct which requires a copy through memory. Our prototype is currently limited to target functions that return a scalar data type (including pointer types), be-

cause it is generated from an LLVM IR template that requires an accurate declaration of the return type. This limitations could potentially be alleviated by an assembly implementation of the monitor functions, or by a wrapper function for the monitor that is written in assembly.

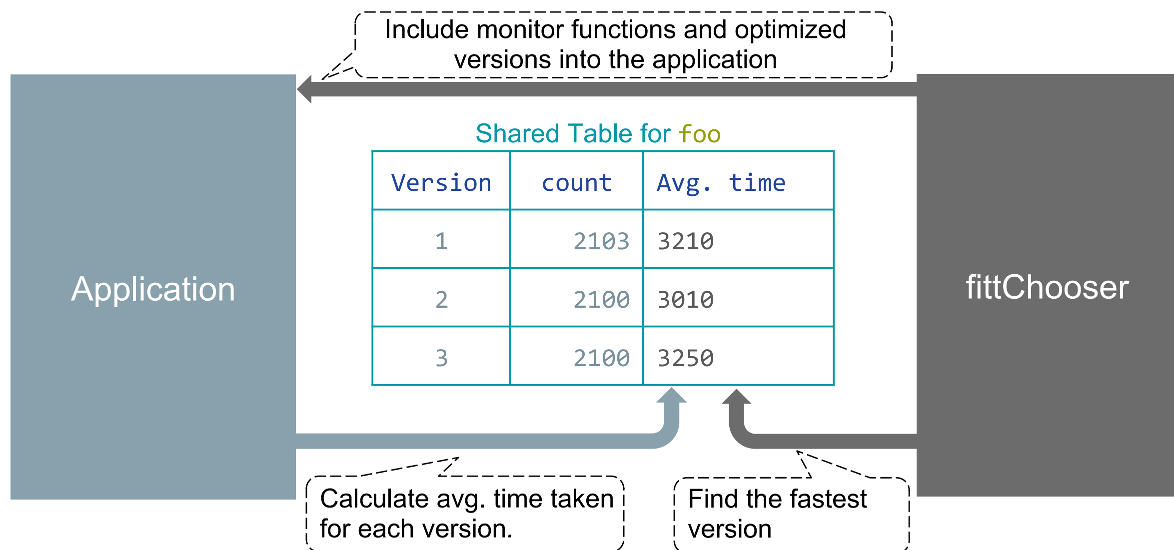


Figure 2.2 – The shared Monitoring table is accessed by both the application and FITTCHOOSER.

2.3.3 FITTLAUNCHER

The user may prefer to conduct an optimization discovery phase and later incorporate those optimizations without running FITTCHOOSER. For this scenario we provide the FITTLAUNCHER, which is an extension of the Linux loader that links a set of predefined function variations into the program at load time. Since this deployment model operates in-process and only takes action at module load time, it eliminates the separate FITTCHOOSER process along with its overhead. This brings additional advantages such as compatibility with GDB—the Linux `ptrace` API only supports one connection at a time, but both GDB and FITTCHOOSER need to communicate with the target program over `ptrace`. There is also potential complexity for the user associated with FITTCHOOSER since it requires configuration of thresholds and other special knowledge, and this complexity is greatly reduced with FITTLAUNCHER once the desired optimizations have been chosen.

This two-phase approach with FITTLAUNCHER also opens a door to more aggressive optimizations in FITTCHOOSER. For important programs that warrant a dedicated optimization effort, the user may conduct an exploratory phase in which FITTCHOOSER is configured to take greater risks. The performance of these exploratory runs may be very poor over all—since many of the attempted variations will be unsuccessful—but FITTCHOOSER will be able to evaluate a broader range of candidates that may lead to discovery of unexpectedly effective variations. After configuring the FITTLAUNCHER to incorporate the best performers into the program at load time, the program can benefit from the speedup without any further overhead from FITTCHOOSER.

The FITTLAUNCHER can either be installed in the operating system to take effect for all programs, or it can be invoked selectively by passing the name of the program to launch along with its arguments (the default Linux loader supports the same usage model). As FITTLAUNCHER loads program modules into memory, it consults a database of installed optimizations. If any are found, it invokes `mmap` to request a region of memory near the corresponding module and populates it with the optimized function variations. Then FITTLAUNCHER links each function by inserting a 5-byte hook in the prologue of the original. To eliminate overhead from the hook, a more advanced implementation could identify the function callers and simply change their target operand, as in FITTCHOOSER. But our experimental results show that even with the hook, FITTLAUNCHER imposes less than 0.2% overhead (geometric mean) across the SPEC 2017 benchmark suite [SPE].

To maintain compatibility with the host Linux platform, we provide a Python script to generate the FITTLAUNCHER from the existing system default loader. The script adds an executable section to the end of the loader and installs a callback hook in the main executable section where internal accounting is performed for a newly loaded module. We implement the FITTLAUNCHER functionality as a static library in plain `C` and splice it into the appended executable section of the loader. To avoid dependencies on loaded modules (which are generally not available to the loader itself!) the FITTLAUNCHER generator script identifies useful symbols such as `open` (for opening files) and `strcpy` in the original loader and statically links them to the FITTLAUNCHER internal functions as necessary.

2.4 Results

We conducted our performance experiments on a 2.7GHz Intel Core i7 Broadwell desktop supporting SIMD and AVX2 with an L3 cache of 4MB and 16GB RAM. The machine runs Linux 3.19 and our benchmarks are compiled with LLVM version 3.7.0 at level `-O3`. We use `taskset` to pin the application to a single core and the `time` command for measurement.

Our experiments focus on a subset of benchmarks from PolyBench [POL] and *SPEC CPU 2006* [Hen06] benchmark suites. The subset is partly necessary because Padrone only tested for programs written in `C`. More importantly, our main goals in this evaluation are to (a) show the potential speedups that FITTCHOOSER can discover, and (b) to demonstrate that FITTCHOOSER can apply these optimizations efficiently, without squandering the speedup. For many benchmarks in these two suites, there is either no function compatible with FITTCHOOSER (for example because of a complex return type), or the benchmark contains no candidate functions for our limited repertoire of program transformations. So we focus our experiments on benchmarks having candidate critical functions under the proposition that future versions of our tool will be able to successfully optimize the excluded benchmarks using an expanded repertoire and a few technical improvements.

We make two minor adjustments to The PolyBench suite because it calls the critical function only once, whereas FITTCHOOSER is designed to optimize iterative programs. Our changes include (1) a `for` loop to call the function one million times and (2) `__attribute__((noinline))` to prevent inlining of the critical function.

2.4.1 Overhead

Although there is significant computational overhead for both the Initial Profiling and Optimizaton Pass, the majority of the overhead is masked by performing the processor-intensive tasks in the parallel FITTCHOOSER process. Upon reaching Cruise Control the periodic profiling has negligible overhead because it is invoked sparsely and for a short duration. Figure 2.3 shows the overhead of (a) profiling alone and (b) profiling and monitoring combined across four benchmarks from the PolyBench suite. The configuration for profiling includes 3 sessions of 5, 10 and 20 seconds with a frequency of 100Hz, 200Hz and 400Hz, respectively. Monitoring is configured to terminate at a

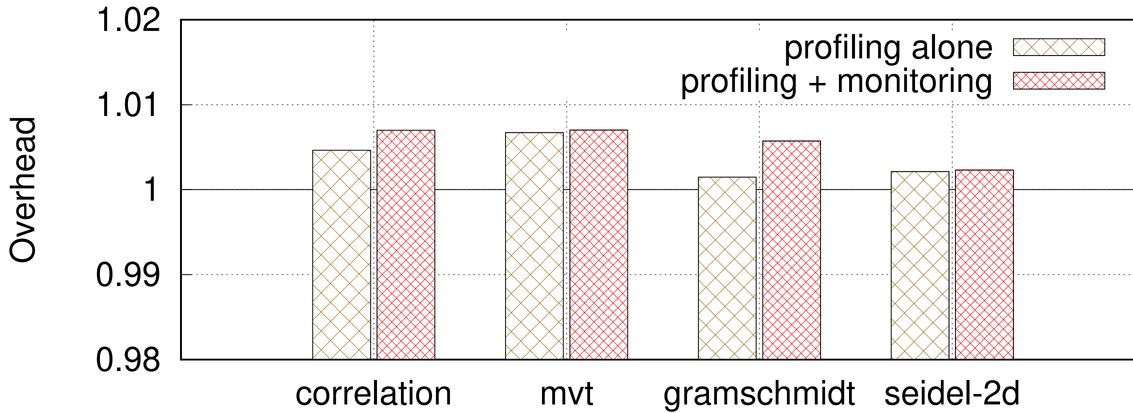


Figure 2.3 – Overhead of FITTCHOOSER.

threshold of 20,000 total invocations of the critical function, and deploys a *null optimization* which simply contains a copy of the original critical function (compilation time of the copy is included in these results). This represents a median scenario where the attempted variations collectively perform roughly the same as the original. Performance can deteriorate if more aggressive (and less reliable) optimizations are attempted. As shown in the figure, the overhead is less than 1% throughout the course of the benchmark in all 4 cases.

The overhead of profiling is independent of the number of critical functions. However, the monitoring phase overhead may vary depends on the number of critical functions because the FITTCHOOSER injects one monitor function for each critical function. The more the number of critical functions more will be the overhead created in the monitoring stage. The number of critical functions that are suitable for the optimization in an application may vary in each invocation of it. For example, there may be some functions in the application which execute only when a specific input flag is enabled. So it is difficult to say an upper limit for the overhead in an application. Figure 2.3 shows overhead when the maximum number of critical function is fixed to be only one. This figure can be used as just a reference for the possible overhead.

2.4.2 Speedup

Figure 2.4 reports the overall speedup obtained for the selected subset of the PolyBench and SPEC CPU 2006 benchmark suites. This includes the Initial Profiling and

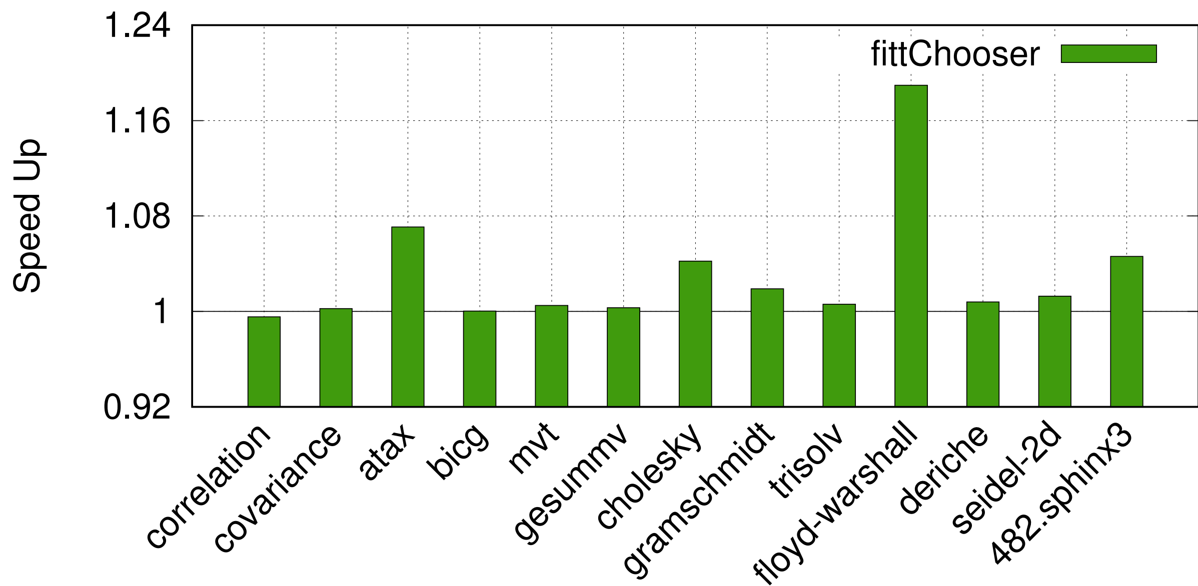


Figure 2.4 – Overall speedup under FITTCHOOSER.

the full Optimization Pass with all associated overheads. Both the benchmarks and the injected variations are compiled at LLVM optimization level `-O3`. The configuration attempts 13 variations of the top critical function in each application. The first variation is produced by recompiling the IR with only the `march=native` flag. The remaining 12 variations progressively assign the `-loop-unroll` flag from 2 to 24 (stepping by 2). In the Optimization Pass, each variation is invoked at least 100 times with a quanta of 10, and timing is measured over the last 100 invocations after discarding the highest 10 results to compensate for noise. Figure 2.5 depicts the performance of these optimization flags on four PolyBench programs.

We experienced a slight slowdown for the `correlation` benchmark. It showed an overhead of 0.6 % in Figure 2.3 and slowdown of 0.4 % in Figure 2.4, indicating that the speedup created by the optimized versions did not recover the overhead of the trials. Table 2.2 shows the standard deviation of the results. While the majority of the benchmarks cannot be improved beyond the `-O3` optimization level, several benefit greatly from FITTCHOOSER: `floyd_warshall` is 19 % faster, `atax` 7 % and `cholesky` 4 %.

Figure 2.5 shows that FITTCHOOSER selects different optimizations for different programs. For example, `kernel_floyd_warshall` performs best with a loop unrolling of 14

Benchmark	Standard Deviation
correlation	0.49%
covariance	0.25%
atax	0.15%
bicg	0.01%
mvt	0.13%
gesummv	0.02%
cholesky	0.05%
gramschmidt	0.23%
trisolv	0.01%
floyd-warshall	0.07%
deriche	1.20%
seidel-2d	0.01%

Table 2.2 – Standard Deviation

whereas `kernel_cholesky` prefers 12. In cases where the performance is almost equal for all unrolling factors, FITTCHOOSER may assign a different variation depending on the exact performance observed during the execution. For example, among 10 executions of `kernel_floyd_warshall`, FITTCHOOSER assigns the 8th version six times, the 1st version three times and the 7th version once.

We also observe that some applications are drastically improved simply by recompilation on the target machine using the default `-O3` optimizations. This still indicates an advantage of FITTCHOOSER over the conventional software distribution model where compilation is performed once at the vendor's site (similarly to [Nuz+13]). No matter how trivial or sophisticated the source of the speedup, the dynamic optimization model finds increasing importance in today's rapidly expanding landscape of computing resources. Applications designed for desktop computers are commonly run on cloud servers, virtualization platforms and even mobile devices, introducing performance characteristics that can vary dramatically from the machine where the code was compiled. To the best of our knowledge, the only way to reliably tune application performance in such an environment is to optimize at the point of execution, which is where FITTCHOOSER excels.

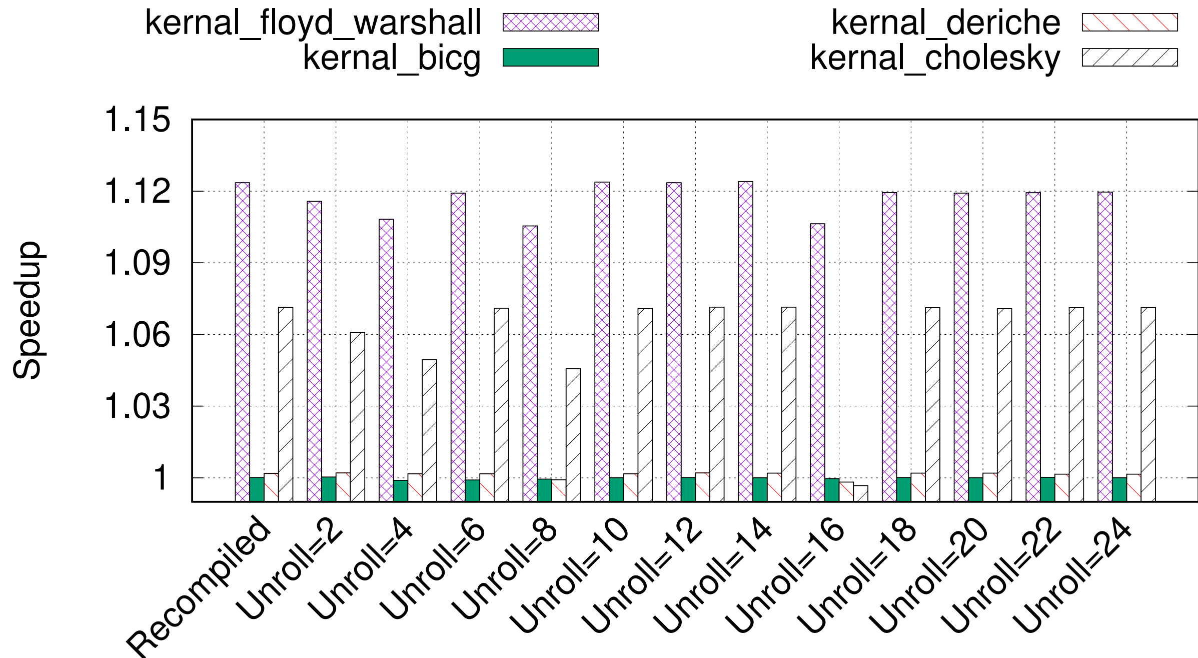


Figure 2.5 – Performance of critical function variations.

2.4.3 FITTLAUNCHER

Since the expected usage of FITTLAUNCHER is to install optimizations for the program's most critical functions, we prepare our evaluation of FITTLAUNCHER by installing a *null optimization* for the top critical function of each program in the SPEC 2017 benchmark suite (as reported by Linux `perf`). We observe a geometric mean of 0.124% overhead, which falls below the standard deviation of 0.178% across the native executions of the suite.

2.5 Related Work

Dynamic Binary Rewriting Pin [Luk+05] is a dynamic binary instrumentation framework with a flexible API that has enabled development of a rich set of Pintools for architecture exploration, emulation and security. Because Pin focuses on instrumentation and analysis, it always runs the target program from a copy in its code cache. DynamoRIO [Bru04] is a similar tool that focuses on efficiency and provides a simple lightweight API to clients. It can execute the target program entirely from its code cache, or partly native, and can consolidate cached code into traces for efficiency. Val-

grind [NS07] focuses more on its instrumentation capabilities than performance, and the framework is designed for *heavyweight* tools: every instruction is instrumented, and a high volume of information about the target program's execution is collected. The novelty of Valgrind is the use of *shadow values* [Net04] for register and memory locations, yielding a more powerful analysis at the cost of higher overhead.

Iterative Compilation is similar to FITTCHOOSER in that it addresses the performance issues that arise from detailed hardware characteristics and transitory factors of the runtime environment. The key idea is to identify local minima by producing many versions of the same program and running them on various platforms to identify the best overall performers. Iterative compilation has been advanced by machine learning techniques that are broadly covered in a survey by Ashouri et al. [Ash+18]. Our work takes the same basic approach, but we apply it at runtime. By doing so, we concentrate only on the most time consuming functions, and we can easily adjust our optimizations for the performance characteristics that are directly affecting the current execution of the program.

JIT Compilers apply different levels of optimizations to functions when they become time consuming (see for example the discussion of Oracle's HotSpot compiler [PVC01]). Their purpose is different from ours: they want to spend time optimizing functions only when the chances are high to recoup the time in future execution time. Optimizations available in each level are fixed, while we explore many variants.

JIT technology with C/C++: Feedback-directed dynamic recompilation for statically compiled languages [Nuz+13] discusses about dynamic optimization by using both native executable file and intermediate representation (IR) file of the program. During execution, they recompile hot methods from IR file using a Java JIT compiler and the recompiled versions are stored in a *Code Cache*. With the help of a trampoline created at the beginning of the original function body, the function calls are redirected to the new recompiled version. This work is very close to our work. We both use almost similar recompilation technique to create optimized version. However this work creates only one version of the function and believes that version is better than the original one present in the binary without taking any feedback about its performance.

2.6 Conclusion

Detailed information about hardware performance characteristics can improve compiler optimizations, but applications are typically compiled for use on many different architectures having a broad range of performance characteristics. Transitory factors of the runtime environment can also affect application performance. We propose FITTCHOOSER to dynamically evaluate the fitness of candidate optimizations for a program's critical functions and then replace the original functions on the fly, all without restarting the program. Experimental evaluation of FITTCHOOSER on important industry benchmarks demonstrates up to 19% speedup even with a limited repertoire of program transformations, suggesting even more gains may be possible as more sophisticated optimization techniques are incorporated into FITTCHOOSER.

CHAPTER 3

OFSPER: ONLINE FUNCTION SPECIALIZER

3.1 Introduction

Function Specialization (also known as *Procedure Cloning*) is one of the optimization techniques applied to the functions in a program based on its parameters [BGS94]. In this technique, different versions of a function are created according to the most frequent values taken by its parameters. In the case of function specialization, it is also often difficult to predict/know during the static compilation phase the argument value/behavior.

To create specialized versions of a function at static compilation phase, the compiler needs to assume or predict some values or some common behavior to the parameters which might not be feasible in many cases. But that is not the case with dynamic optimization where the actual values or behavior of arguments are known. We have developed a dynamic/online optimization tool called OFSPER which can apply function specialization at run time. OFSPER creates specialized versions of the function, according to the actual value of parameters, during the execution of the process.

A more detailed explanation of function specialization and its scopes are given in Section 3.2. Section 3.3 gives a general idea about dynamic function specialization and a detailed explanation of our implementation is provided in Section 3.4. Section 3.5 illustrates our approach with a simple example. Section 3.6 shows the experimental setups and the impact of OFSPER on different benchmarks. And finally, Section 3.7 discusses related work, and Section 3.8 concludes our work.

3.2 Function Specialization

Function specialization is one of the optimization techniques used to reduce the execution time of a function. The idea is that instead of calling a generic function for all the call sites, call different versions of it according to the values taken by the parameters. The call sites of a function are divided into groups based on the values taken by the parameters, and a specialized version of the function is produced for each group [BGS94]. Each version is specially optimized for one or particular category of arguments so that they are expected to run faster as compared to the original generic version for such arguments. For example, consider the function Foo in Listing 9. All call sites of Foo, where value of parameter b is a power of 2, can be considered as one group and a specialized version Foo_b2n can be produced for it. Now, Foo_b2n can execute faster compared to Foo when $b = 2^n$, because of the use of shift operator instead of more expensive division operator used in Foo. So the call Foo(a,8) can then be replaced by a call to Foo_b2n as Foo_b2n(a,3).

Listing 9 Function Specialization

```
Foo(a, 8);
double Foo(unsigned int a[], unsigned int b)
    for i = 1 to  100
        c += a[i]/b;
return c;
        (a) Original code
  ----------------------------------

Foo_b2n(a,3);
double Foo_b2n(unsigned int a[], int n)
    for i = 1 to  100
        c += a[i]>>n;
return c;
 (b) Specialized code when b = pow(2,n).
```

For applying function specialization, knowing the value of the parameter(s) is the key. In most of the cases, the function calls do not contain constants as arguments, instead they have variables, like in Foo(a,x). In such cases, it is not straight forward to do function specialization since the values of variables might be unknown. With a profile-guided optimization, having a simulated execution of the code during compilation for knowing the behaviour of the program, the value or property of the parameters can be predicted. However it may not be a feasible solution all the time because the pre-

dicted behaviour can vary at actual run time. Therefore, applying function specialization during static compilation phase is very difficult, which essentially means specialization would be more effective when applied dynamically by knowing the exact values of variables.

Knowing the value of a variable provides scope for various optimizations, such as *constant propagation, constant folding, algebraic simplification, strength reduction, unreachable code elimination, short circuiting, loop unrolling, vectorization* etc. [Muc97; BGS94]. Optimizations like *constant propagation, constant folding* and *algebraic simplification* allows an expression in the program to be precomputed, which can speed up the execution of the program. Similarly, expensive operators could be replaced by equivalent less expensive operators by applying *strength reduction*. In some cases where the trip count of loops depends on the parameter value, the trip count can be precalculated and so *Loop unrolling* and *vectorization* could be more effective.

Such repetition of values may be surprising at first glance. Analyzing the reasons behind this behavior is beyond the scope of this paper. However, we note that it has been observed before, and this is not the sign of poor software or compiler. Modular software engineering and code reuse contribute to this phenomenon, as well as underlying semantics of the data being processed (modeling the real world).

3.3 Dynamic Function Specialization

Dynamic function specialization is a program optimization technique in which function specialization is applied on a running application to improve its execution time. In this technique the different versions of the function are created dynamically according to the live values taken by its parameters. Since knowing the actual value of arguments is very important to perform function specialization, it will be more effective if applied on a program in execution. Further, a more hardware specific optimized version can be produced in this technique due to the knowledge of running hardware platform, compared to static function specialization technique.

3.3.1 Use case

Dynamic function specialization is useful when the static compiler is unable to extract/identify the values taken by the parameters of a function to create the specialized versions statically. A specialized version of a function is needed when there is a good chance of calling the specialized version. That is, the probability of repeating at least one of its parameters should be high. Some functions do repeat their arguments, but not all the time. We monitored some of the critical functions in *SPEC CPU 2006* benchmark suite [Hen06] and captured the values taken by their parameters. The result obtained is interesting and is shown in Table 3.1. Some of the parameters are taking the same value across multiple function calls. Moreover, the idea of memoization presented in [Sur+15] is entirely based on functions with repeating arguments and they have listed more such functions. These all are indicating the possibilities of applying dynamic specialization.

Benchmark	Function	No. of calls [1]	Time (%)[1]	Unique Values
sphinx3	vector_gautbl_eval_logs3	2.3 M	26.83	1
hmmer	FChoose	277.2 M	1.88	1
sphinx3	subvq_mgau_shortlist	492.9 M	8.17	1
mcf	primal_bea_mpp	2.2 G	40.29	2

[1]`gprof` data

Table 3.1 – Repeatability of arguments

3.3.2 Our Approach

A function may be called from different parts of the program. Since the values taken by the parameters may differ in each call even from the same call site, it is not possible to directly replace the original function call by a call to a specific specialized version. Instead, different versions need to be maintained according to the arguments. In our approach, we use an extra function, called *monitor function*, to manage these specialized versions and redirect function calls to appropriate versions. The *monitor function*s, one for each function, are created dynamically and we replace all the original function calls by a call to the *monitor function*. Figure 3.1 shows the difference in function execution before and after applying specialization. In normal execution of the program, the original function, `Foo`, is directly executed. But in the case of dynamic specialization the *monitor function*, `Foo_monitor`, is executed first and then the appropriate version is

called from it.

Since the *monitor function* is not part of the original application, it creates some execution overhead on the application during each call to the original function. Dynamic function specialization can be beneficial only when the optimized versions gain enough speedup to overcome the overhead created by monitor function. Hence, before creating the specialized versions, repeatability of the arguments must be ensured so that the optimized versions are expected to be executed more number of times compared to the original function. The values taken by the parameters of a function are observed initially for ensuring repetition before creating specialized versions. Specialized versions are created for such repeating arguments in parallel to the execution of the program.

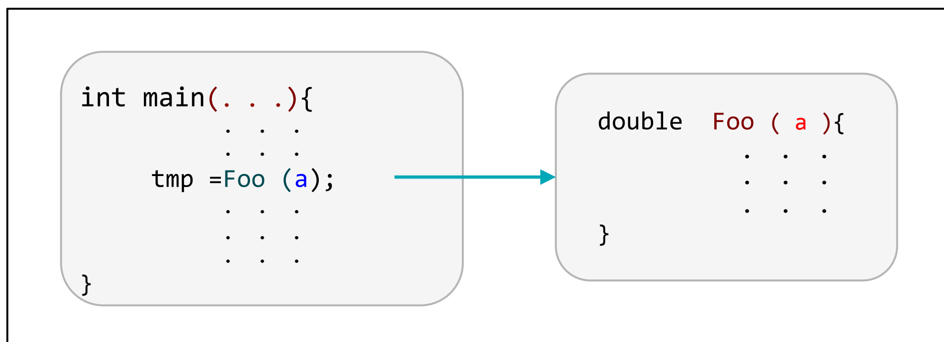
The overall procedure is carried out in two phases, *Analyzing and Monitoring phase* and *Specialization phase*. Analyzing and monitoring phase is the decision-making phase in which the functions which are needed to be specialized are determined. In specialization phase, the different versions are created and included into the process. We now explain these two phases in detail.

Listing 10

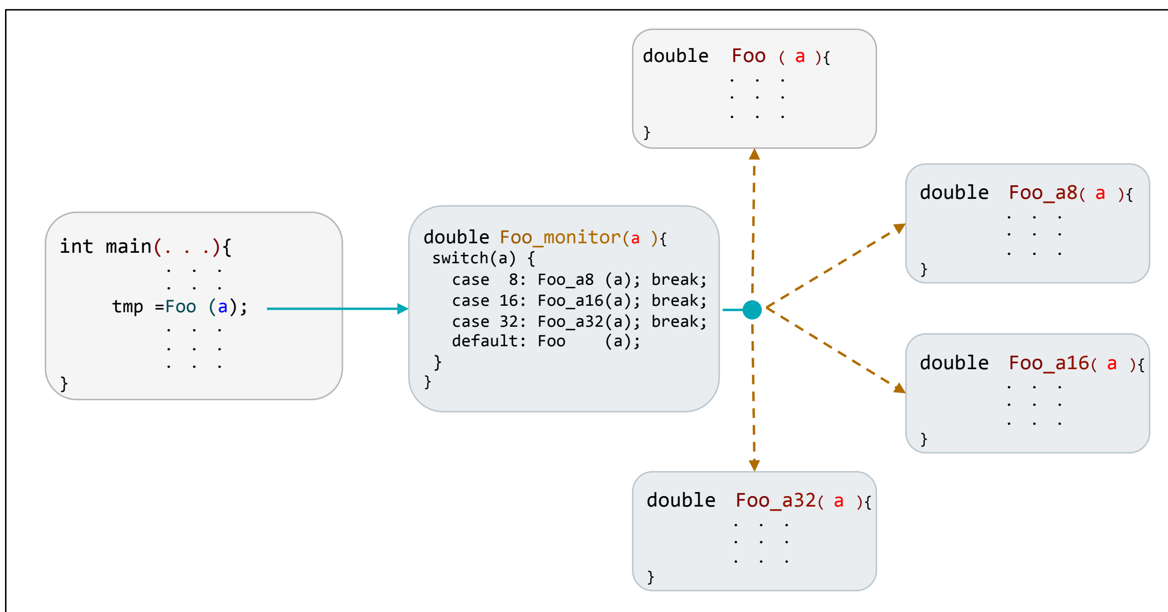
```
double Sum(int a[], int b)
    for i = 1 to  b
        c += a[i];
return c;
```

Analyzing and Monitoring phase

The first step in this phase is to find out the suitable 'hot' functions for specialization. The hot functions in the application can be determined by monitoring the spots where the application spends most of its CPU time. Many techniques have been proposed, such detection can be performed with very low overhead [Rio+14]. Not all hot functions may be suitable for specialization. Suitable functions for specialization are chosen based on how the parameters are used inside the function body. If knowing the value of the parameter creates new possibility for applying different optimization techniques, like those discussed in section 3.2, then the function can be marked as suitable. For example, consider the function `Sum` given in Listing 10. In this case the trip count of the loop can be precomputed once the value of `b` is known. And by knowing the trip



(a) Normal Execution.



(b) Dynamic Function Specialization

All calls to function `Foo` are redirected to `Foo_monitor` and `Foo_monitor` decides which version to be executed

Figure 3.1 – Call sequence: Normal vs Specialization

count different optimization techniques can be applied efficiently [BGS94; DH79]. More details on choosing suitable function are given in Section 3.4.

The next step is to collect the actual values taken by the optimizable parameters of the functions. After a function is found suitable for specialization, a *monitor function* is created for it and all calls to it are redirected to this *monitor function*. The *monitor function* is used to collect arguments. It contains a table, as shown in Table 3.2, which can store arguments, their repetition count and target function indicator. On every call to the function, depending on the argument, repetition count is incremented and corresponding target function is called. Initially all the entries of the *Target Function* column are set to the original function. The need of applying specialization is decided by the repetition count of the argument. If none of the arguments is repeating for a given amount of time, monitoring is disabled by restoring the original function calls. It can be re-enabled at a later time to capture a change in application phases.

Argument Value	Repetition Count	Target Function
8	3128	Foo
13	129	Foo
16	2451	Foo

Table 3.2 – Monitoring Table

Specialization phase

The specialized versions of the function are created based on the repetition count of the arguments. When the count of any of the arguments reaches a threshold value, a specialized version is created. We currently rely on the availability in the program executable file of an intermediate representation of the program generated during the compilation. The technique is sometimes referred to as *fat binaries* (see for example the recent work of Nuzman et al. [Nuz+13]). The specialized version is produced by recompiling the intermediate representation after replacing the corresponding parameters with their values. The compiler can then apply constant propagation followed by all available optimization techniques, including hardware-specific optimizations, according to the parameter value.

The optimized versions are included in the process by injecting their binary code into the process memory space. The *Target Function* value in the corresponding table entry

is then modified so that future calls to the function with this argument will execute the specialized version. For example, if we consider 1000 as the threshold value, then there is a possibility of making two specialized versions of `Foo` based on Table 3.2 entries. The specialized versions `Foo_b8` and `Foo_b16` can be created for the values `8` and `16` respectively and then the table entries are modified as in Table 3.3.

Argument Value	Repetition Count	Target Function
8	3128	Foo_b8
13	129	Foo
16	2451	Foo_b16

Table 3.3 – Modified Monitoring Table

Special case In the case of pure functions, such as the transcendental functions like `sin`, `cos`, `log` etc, by knowing the value of the parameters, we can directly calculate and store the result of the function as these functions are known to return the same value across calls for the same argument(s). In such cases, it is not required to create a specialized version and instead we just need to store the result. This type of specialization is known as *function memoization* [Sur+15]. We consider such functions separately and create a special kind of *monitor function* with a separate table structure. In this table, we store only arguments and results as shown in Table 3.4. For each argument, *monitor function* executes the original function on first call and stores the result in the table and returns this result for subsequent calls with the same argument.

Argument Value	Result
5	32
8	256
16	65536

Table 3.4 – Monitoring table for the pure function `exp2`

3.4 Implementation Details

3.4.1 Overview

Our specialization approach includes four major tasks.

Profile: Find out ‘hot’ functions of the application.

Analyze: Choose ‘hot’ functions which are suitable for specialization.

Monitor: Collect values taken by the parameters of suitable functions.

Specialize: Create specialized versions of the function for repeating arguments and include them into the application.

The first three tasks are part of *analyzing and monitoring* phase and the last one is of *specialization* phase.

These tasks are performed by OFSPER which runs in parallel with the target application process. The target program remains unmodified, and does not even need to be restarted. OFSPER operates in a manner similar to a debugger, attaching to and detaching from its target. More details are given in Section 3.4.2.

To achieve our goal of dynamic function specialization, we need to make some changes to the application’s memory space, like changing function call instructions and including new binary codes of the functions. Like FITTCHOOSER, OFSPER also use Padrone for profiling, for injecting new functions and for the redirection of function calls.

Since the available information in machine-level code is very limited, optimizing a binary code alone is very difficult. So we use, similar to [Nuz+13], both LLVM intermediate representation [LA04] (LLVM IR) and binary code of the program. LLVM IR is used for creating optimized versions of the functions and the binary code is for executing the program. The LLVM IR is produced from the source code during the compilation of the program.

Although our current approach relies on fat binaries, we plan to drop this requirement by lifting binary code to LLVM IR. Previous work by Hallou et al. [HRC16] using the McSema infrastructure [DR] showed that this is a viable path, including for optimizations as complex as vectorization.

3.4.2 OFSPER

We implement the idea of dynamic function specialization with the help of our OFSPER tool which runs alongside the application as shown in Figure 3.2. The specialization process is carried out in this separate process (thanks to PADRONE) to reduce the

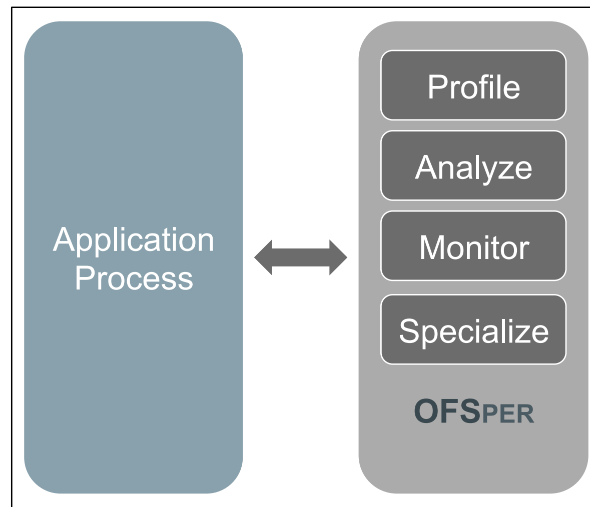


Figure 3.2 – OFSPER attached to the application process

impact on the target application (at least on a multicore, or a processor equipped with simultaneous multithreading such as Intel's HyperThreading).

The optimized versions of a function are created from the LLVM IR of the program produced during the compilation of the source code. We write LLVM $passes$ [Llv16; LA04] for creating optimized versions and *monitor function* of a function and also for finding the suitable functions for specialization among the critical ones. We use three different passes.

isPossible pass: Used in *analyzing* stage to check the suitability of a function for specialization.

monitor pass: Used in *monitoring* stage to create *monitor function*

optimize pass: Used in *specialization* stage to create specialized versions of a function.

OFSPER starts its execution just after the application process starts running. Then, PADRONE is used to attach it to the application process. The detailed explanation of each stage of its execution is given below.

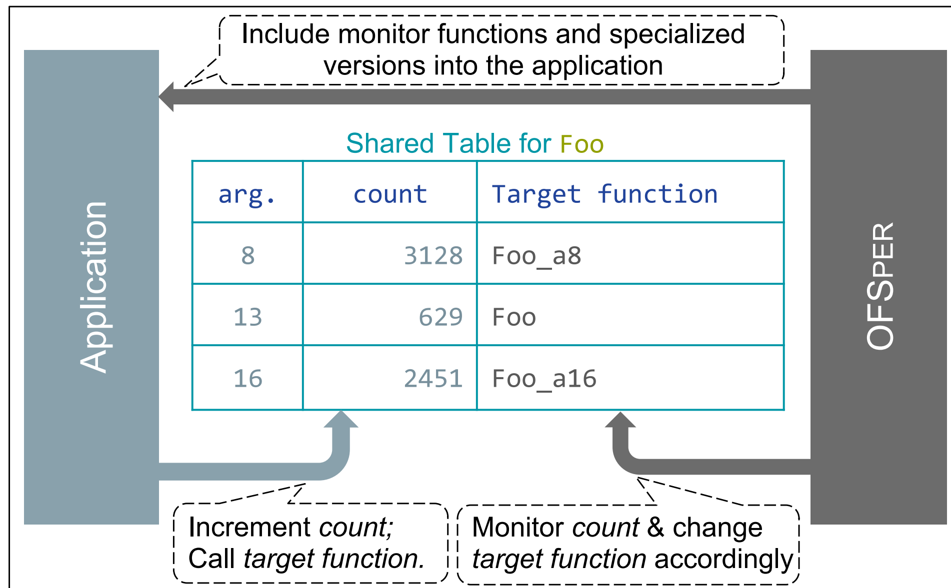


Figure 3.3 – The shared monitoring table is accessed by both the application and OFSPER

Profiling Stage

Instead of specializing every function in an application blindly, we apply specialization only on critical functions in it. OFSPER use profiling, a feature provided by PADRONE, to find out these critical functions. PADRONE probes the performance counters of the processor at regular interval of time with the help of *Linux perf event* subsystem kernel interface. Each probe provides a sample and we use these samples to figure out the critical functions by inspecting the instruction pointers included in each sample. One session of profiling lasts only a few seconds and it repeats at regular intervals of time to catch more live critical functions.

Analyzing Stage

All functions may not be suitable for specialization. Functions with no parameters may only benefit from a hardware-specific optimization, but not from specialization. Since our *monitor function* creates some overhead, the specializable parameter of the function need to be used in a critical part of the function code such that specialization should not result in a slow down to the overall process. Our cost-model attempts to capture all these phenomena. Since, loop optimizations heavily impact the performance of a

program, a parameter which is part of the loops can be a good candidate for specialization. Parameters of pure functions are also considered since the entire function call can be reduced to a constant.

The *isPossible* pass is used to find out functions which are suitable for specialization. In this pass, we analyze each uses of integer or floating point type parameters of the function. Once we find a use in an expression calculating trip count of a loop, we backtrace the uses of other operands in the expression. If the other operands of the expression are derived either from integer or floating point type arguments or from constants, then we mark the function as specializable. Currently we are looking only for arguments of `int, long int, float` and `double` data types but this can be extended to other data types. This pass outputs the name and data types of the parameters which satisfy our condition and a flag indicating whether the function returns `void` or not. The list of all functions which are suitable for specialization could also be made earlier, during creation of the LLVM IR of the program, and can then be used at run time to pick out specializable critical functions. This would minimize the overhead, but we have not explored that direction yet.

Monitoring Stage

For each suitable function, we create a *monitor function* for collecting arguments and redirecting function calls to appropriate versions. The arguments are stored in a look up table inside *monitor function*, as in Table 3.2, to detect the possibility of specialization. The *monitor function* performs two operations.

1. Increment *repetition count* corresponding to the argument
2. Call the corresponding *Target function* and return its result.

The table is hosted in a System V shared memory segment, and shared between the two processes as shown in Figure 3.3. The application process, through *monitor function*, updates the first two columns of the table while the third column is modified by the *optimizer* process. Initially the *Target Function* column of all the table entries are set to original function and they are modified accordingly on creation of each specialized version. A hash function is used to index the table. In order to perform a quick table look up, we use the folding `XOR` based indexing as used in [Sur+15]. The idea is, higher

order and lower order bits of the argument are repeatedly `XOR`ed until we get a 16 bit number. And we mask 4 bits of it to get a 12 bit number which can be used as index to a table of size 2^{12}. In case of a conflict, we directly call the original function without modifying the table entry. For functions with more than one (specializable) argument, first we `XOR` all arguments to a single one and then applies the above procedure on it.

After the LLVM IR of *monitor* function is created, it is compiled to a shared library which can be dynamically linked. Then, with the help of PADRONE this shared library is injected into the process and subsequent calls to the original function are trapped and redirected to the monitor function. Now on, all the values taken by the specializable parameters are collected in the table inside *monitor* function which is used in *specialization* stage.

Specialization Stage

The need of specialization is decided by looking the values taken by the parameters of the function so far. The values and their repetition counts are stored in shared tables. These tables are analyzed by optimizer process at regular intervals. A specialized version of a function is created for a particular argument, when the repetition count of that argument crosses a threshold value. We set the threshold value for a function to 10% of its total number of calls so far or to 500, whichever is greater.

The *optimize* pass is used to create the LLVM IR of the specialized version from the LLVM IR of the program. This pass makes a copy of the original function with the name of specialized version. Then all the uses of specializable arguments are replaced with their exact value. The resulting LLVM IR is then compiled to get the executable file of the specialized version. The compilers apply all the suitable optimization techniques based on the argument value. It is also possible to create more hardware specific versions from the LLVM IR [Hal+15]. The specialized version is then injected to the process using PADRONE and the value of *Target Function* in the corresponding table entry is changed to the specialized one as shown in Table 3.3.

If none of the arguments are found repeating more than the threshold value even after the function is executed more than 50k times, then the function is removed from the suitable functions list and it will not be considered for monitoring anymore.

Listing 11 A specializable function

```
long int foo(int a[], int b[], int p, int q)
    long int i, j, k=0;
    for j = 0 to (p+q)*2
        for i= 1 to q*2
            k+= ((a[i]/b[i])) % 2000;
            k = k/(p*q-9998*q-2047);
return k;
```

We repeat these four stages until the application finishes its execution. In some cases where the function has more than one suitable argument, it is also possible to have only a subset of them to be repeating. In such cases, we may need to create the *monitor* function once more by considering only the repeating arguments.

Additional benefit

As mentioned earlier, a compiler can apply more optimizations to a program, once it knows the hardware details of the running machine. Our specialized versions are optimized based on the hardware. However, before creating specialized versions, we execute the original function. So it is beneficial to produce also a hardware-specific version of the original, non-specialized, function.

Handling pure functions

The library functions, like `cos`, `sin`, `exp` etc, are considered separately. In such cases, we make only *monitor function* and it can store the result in the table. So, we do not need to create specialized versions. To redirect the original function calls, PADRONE updates the GOT table entry with the address of *monitor function*s. We need to execute the original function for the first call of each argument value to get the result. Since the GOT table entry is modified, we cannot directly call the function by its name from the *monitor function*. Therefore we use `dlsym` to fetch the address of the function and we call the function by its address.

3.5 Example

This section illustrates how our optimizer process may impact the execution time of a function with the help of a simple example. We used the function given in Listing 11. We call it from an infinite loop, and we measured its execution time on each call. Figure 3.4 reports our observation. The x-axis represents the time elapsed and the y-axis represents the execution time (average of five consecutive calls) of the function. Before specialization, it takes around 2.13 seconds to complete the execution with the values 9999 and 2018 to the arguments p and q respectively. At the 180th second, we started a fairly aggressive profiling period for 100 seconds with a frequency of 1000 samples per second. This is visible as a small bump on the graph. During this profiling session, the optimizer process identifies *foo* function as a hot function and starts monitoring its parameters. The monitoring continued until the repetition count reaches the threshold. At around the 730th second the count crosses the threshold and a specialized version of *foo* is created. The execution time drops to around 2.04 seconds only to complete its execution. Figure 3.4 shows a slight increase in the execution time during profiling and monitoring stages, but on the long run, specialization is clearly able to recoup this overhead.

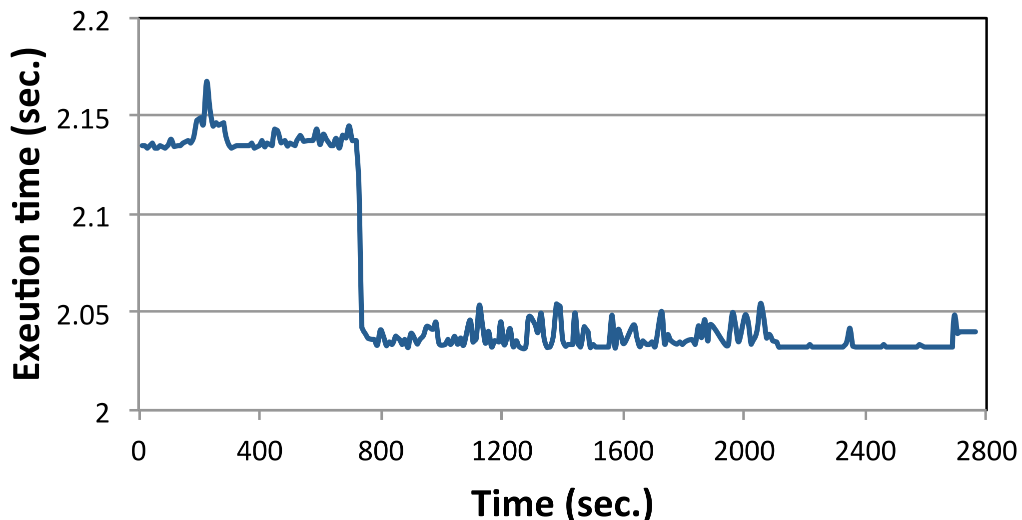


Figure 3.4 – Impact of specialization on execution time of a function (lower is better)

3.6 Result

3.6.1 Experimental set-up

We implemented our idea of dynamic function specialization on an Intel Broadwell core i7 architecture with $4\,MB$ L3 cache and $16\,GB$ RAM. We set the clock speed to $2.7\,GHz$. The LLVM version used is $3.7.0$ with $O3$ optimization on a Linux version 3.19 operating system.

Benchmarks Since PADRONE is tested only for C language programs, currently we implemented dynamic function specialization only for programs written in C language. We choose benchmarks, written in C, which contain functions satisfying our specialization criteria which are discussed in Section 3.4.2. The first one is the function should be critical and it should contains at least one integer or floating point type argument. The second one is this argument should be used in trip count calculation and the argument should repeat. We report only on qualifying benchmarks, since others practically show neither speedup nor slowdown: *hmmer* and *sphinx3* benchmarks from SPEC CPU 2006 benchmark suite, *equake* benchmark from SPEC OMP 2001 benchmark suite [Asl+01] and *ATMI* [Mic+07].

Prerequisites Our optimizer process runs in parallel with the application process. We need both binary executable and LLVM IR of the application. Since all our decisions regarding the specialization are taken at run time, we do not require any prior information about the application.

3.6.2 Overhead

Among the four stages of our implementation, profiling, monitoring and specialization stages directly affect the execution of the application program. For analyzing the slowdown caused by our implementation, we run it in two different situations. In first, both monitoring and specialization stages are disabled and in second, only specialization is disabled. We use *h264ref*, *hmmer*, *sphinx3* and *gobmk* benchmarks from SPEC CPU 2006 for analyzing the overhead. The result is shown in Figure 3.5. All other benchmarks may also perform similarly if there is no specializable functions found. The profiling stage is repeated 3 times. The profiling sessions are carried out for 5 seconds with

100 samples per second, for 10 seconds with 200 samples/second and for 20 seconds with 400 samples/second respectively. And the overhead created by profiling is very less (less than one second). The overhead created by monitor stage depends on the number of times the functions are called. On each function call, we have a table look up and an extra function call to monitor function. So the overhead created by monitor stage is different for different benchmarks. Considering the two extremes in Figure 3.5,

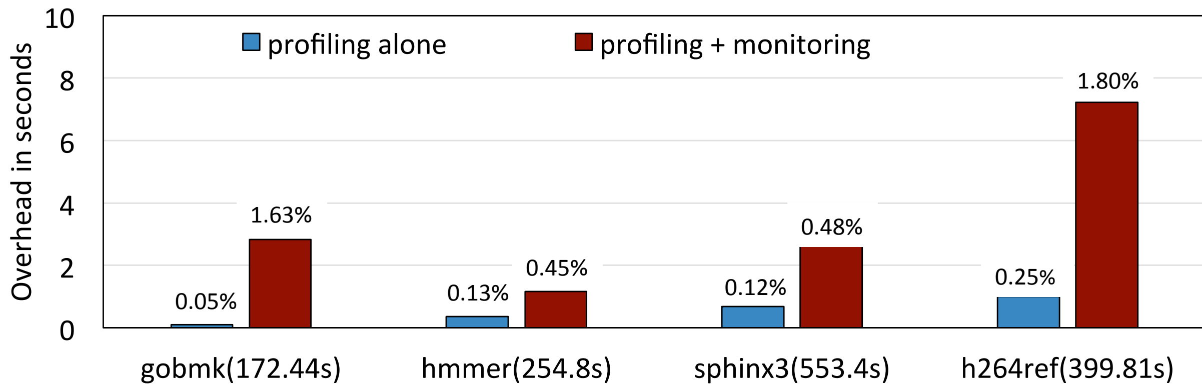


Figure 3.5 – Overhead by profiling and monitoring

Normal execution time of the benchmark (in seconds) is given in brackets and the percentage of overhead is given on top of each bar.

the average number of calls of monitor function in h264ref is around 116 million while for hmmer, it is only 458 thousand. The overhead can be reduced by calling a machine dependent optimized version of the original function instead of the actual one in the case of non repeating arguments. For analyzing the overhead, we used the actual version included in the binary of the program itself.

3.6.3 Speedups

Figure 3.6 reports the speedups we obtained. We are including only the benchmarks in which we can specialize at least one function. All other benchmarks may result similarly in Figure 3.5. The functions `subvq_mgau_shortlist` from *sphinx3* and `primal_bea_mpp` from *mcf* given in Table 3.1 are not specialized because the former one contains a `static` variable and in the later one the repeating argument is not part of the loop. Currently our implementation is not handling functions with static variables.

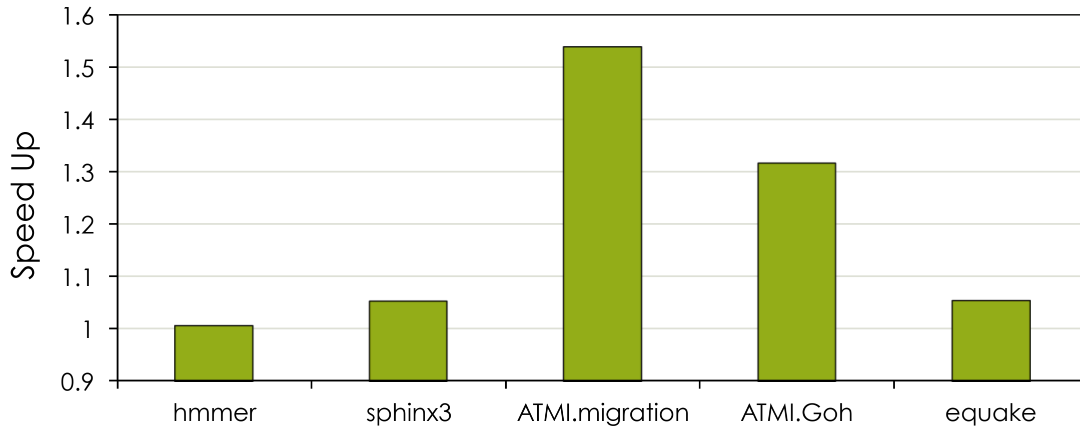


Figure 3.6 – Speedups

The total time taken for the application is measured using `time` command and the speedup shown in Figure 3.6 are measured based on the total time taken by the application. In *hmmer* benchmark, functions *P7Viterbi* and *FChoose* are monitored. But the monitored argument in *P7Viterbi* function is not repeating. In FChoose function, one integer argument is repeating and it takes 20 as its value all the time. The compiler applies *loop unrolling* technique on a loop inside this function to get the optimized version. Since FChoose is taking only 2 % (see Table 3.1) of total execution time of the application, the improvement in this benchmark is minimum. In sphinx3 benchmark, function `vector_gautbl_eval_logs3` is monitored and specialized and we obtain around 5 % improvement at runtime. In this function, two integer arguments are repeating with the values 0 and 4096 on every call to the function. Here, compiler applies *vectorization* technique to improve the run time of the specialized version.

We also obtain good improvements in ATMI application: 35 % and 24 % for examples ATMI.migration and ATMI.Goh respectively. In both cases, we specialized the Bessel functions $j0$ and $j1$. And in equake, we obtain an improvement of 5 % thanks to the sin and cos functions.

3.7 Related Work

This section discuss about a number of works in dynamic function optimization.

Intercepting Functions for Memoization: A Case Study Using Transcendental Functions [Sur+15] discusses about implementing *memoization*, saving the result of execution of a section of program for future use, in software for *pure* functions. The idea is saving the result of a function in a table according to its arguments and return these results, instead of executing the function again, in future calls. The paper shows that a good amount of application have argument repetition in functions even after applying state of the art compiler optimization. But that paper is not studying if any particular value or class of values are being repeated and if we could do optimization based on them. Our work is an extension of this work. We used both memoization in case of mathematical functions and specialization for other functions. Memoization works when the function can be entirely reduced to a constant but specialization can consider intermediate steps: not constant, but some computations are eliminated/simplified.

Just-in-Time Value Specialization [San+13] discusses about creating more optimized version of a function, based on runtime argument value, for JavaScript programs. JavaScript programs are usually distributed in source code format and compiled at the client-side. A just-in-time compiler is used to compile JavaScript function just before it is invoked or while it is being interpreted. In just-in-Time value specialization, they observed that most of the JavaScript functions are called with the same argument set. So they replaced the parameters of the function with the values while compiling at client side to get native code. But, if the function is called with a different argument, the specialized native code is discarded and actual source code is compiled. Then the function is marked so that it won't be considered for the specialization in future. The main difference with our work is that, they are not handling multiple versions of a function. Instead, at a time there is only one version of a function, either the specialized version or the original one. And that version is created directly from the source code.

Tempo [CLM04] Tempo is a specializer tool developer specific to the C language. It provides a declarative language for the developer using which he can provide specialization options for the tool. It has both compile time as well as run-time specialization options. The major difference of this work and ours is that we do not require any help from the programmer and hence our technique can be applied to any existing program. Though due to implementation limits we could only use C programs for our results our

technique is more general and is extendable to any other language.

3.8 Conclusion

Compilers can do better optimization with the knowledge of run-time behaviour of the program. We propose a runtime optimization technique called *dynamic function specialization*. We analyze the values of arguments of a function and create different versions for all candidate functions in parallel with the execution of the program. Then we inject these specialized versions into the running process with the help of PADRONE library. The function calls are redirected to the appropriate versions with the help of an extra function. Our approach does not require restarting the application. Our speedups range from 1 % to 35 % for a mix of SPEC and scientific applications.

Our current implementation relies on *fat binaries* which store the compiler intermediate representation of function in the program executable. Future work will consist in lifting binary code to LLVM IR, opening the door to optimization of any program, including legacy or closed-source.

IMPLEMENTATION DETAILS

This chapter is intended to familiarize the working of the tools with the help of some implementation details. As we mentioned earlier, the tools are using the PADRONE libraries for their operations with the executing program. They include the attaching of the tool with the running application, inserting the binary codes to the application, etc. The other two interesting parts, (a) the format and implementation of monitor functions and (b) the LLVM Passes, are discussed in this chapter.

4.1 Monitor Function

The monitor functions are used to monitor and control different versions of the original function. The monitor function is created especially for each functions. The function arguments and return type of the monitor function should be same as of the original function. For that, we use help of LLVM IR files. Initially, the tools will create a template code of monitor function using the information of the original function acquired from the LLVM IR of the application program. This template code is written in C language and it also contains function declarations of the original function and its optimized versions. This template file is then compiled to get the LLVM IR. After that with the help of a LLVM pass called $monitor\ pass$, which we mentioned in section 3.4.2, the return type and argument list of the original function from the program's LLVM IR is copied to the monitor function's LLVM IR. The working of this LLVM pass will be discussed in section 4.2.

The above mentioned template file contains code for accessing the system V shared memory segment, calculating the version of the function needed to be executed next, calling the appropriate function, recording the result in the shared table and finally

return back to the caller function. Sample codes for the monitor functions used in OFSPER and FITTCHOOSER are given in Appendix B and A respectively.

As we mentioned earlier, the monitor function uses a shared table to communicate with OFSPER and FITTCHOOSER. During the first call to it, it need to map the shared table to the process. For that, it uses a static variable whose value will be changed in the first call so that the subsequent call will not try to map the shared table again. The C preprocessor directives are used to handle the function specific information such as function name, the arguments need to be monitored, return type, the statements for checking the arguments etc. By using these directives, we do not need to write the whole code for each function. Instead, we just need to insert the statements defining these directives in the appropriate place.

Comparison

Both OFSPER and FITTCHOOSER need the help of monitor functions for their operations. However, there is a slight difference between the operations of monitor functions created by these two tools. The monitor function used in the OFSPER is always executed for every function call where as in FITTCHOOSER it need to be executed until finding out the fastest version. The monitor functions created by FITTCHOOSER need to store the return value so that it can calculate the time taken for executing the specific version before returning back to the caller function. However, storing the returned value is not so straight forward if the type of the value is not a basic data type. For example, if it is a `struct` value, the complexity becomes higher. In OFSPER, the monitor function do not need to store the returning value. It can directly return whatever the executed function version is returned to the caller function.

Both OFSPER and FITTCHOOSER use shared tables to communicate with the application. In OFSPER, table entries are stored based on the argument value. Since value of the argument decides the version that needs to be called in the corresponding execution, a hash function on the argument value is used to index the table. It indicates that the size of the table needs to be big in order to reduce the collision. In FITTCHOOSER, the table entries are limited. FITTCHOOSER can decide the number of optimized versions that need to be tested, and these versions are executed in a round robin fashion. This means, there is no need for any hash functions or searching algorithms to find

which version needs to be executed next. The details of the versions are stored in the adjacent table entries. So, the size of the table is exactly equal to the number of optimized versions.

No Need for Locks

The monitor functions should execute as fast as possible to reduce the overhead in execution time of the application process. Since there are shared data between two process, implementing locks seems to be advisable. However, it may increase the execution time in terms of waiting time for the locks. So it will be beneficial if it is possible to avoid the locks.

Even though the shared tables are accessed by both the application process and the tools, there is no need to implement locks to protect the data. In case of FITTCHOOSER, the monitor function updates the average execution time of each version after every execution of the version. FITTCHOOSER need to read this average time after `count` reached a threshold value. Since the calculation of average time is carried out by the monitor function, it does not really matter whether the value read by FITTCHOOSER is from latest execution or from the previous execution. Even if it is from previous execution, then also it is an average time calculated from enough executions.

In OFSPER, the monitor function updates the `count` and reads the value stored in `function`. OFSPER reads `count` and updates `function` when the `count` is more than the threshold value. Initially all the table entries have the same value for `function` which tells to call the original function. This value is changed by OFSPER only when the new specialized version is injected to the process. If the monitor function fails to read the updated `function` value, there is no problem for the application process. It executes the default function as per the initial value.

4.2 LLVM Passes

Both OFSPER and FITTCHOOSER use `LLVM` IR of the application program for doing optimizations. The IR files are analyzed and modified using different `LLVM` passes. The following are some major passes specially written for OFSPER and FITTCHOOSER.

***monitor pass*:** by both OFSPER and FITTCHOOSER to create *monitor function*

***isPossible pass*:** Used by both OFSPER and FITTCHOOSER to confirm the suitability of a function for specialization.

***optimize pass*:** Used by OFSPER in its *specialization* stage to create specialized versions of a function.

monitor Pass

This `LLVM` pass is used to create monitor functions. As we discussed in Section 4.1, this pass is used to make the required changes in the `LLVM` IR of the monitor function created using the template file. As we discussed earlier, the monitor function wants to call the optimized versions and the original copy of the optimizing function. So, the template file contains not only the body of monitor function but also the function declarations of the original function and its optimized versions. The original function and its optimized versions are called for execution from the monitor function. For the proper working, these function declarations and call statements in the template file should be having the same return type and argument list as of original function in the program. Initially, the template file does not match the arguments and return type. This pass is used to make the required changes.

In OFSPER, only the arguments on which the function is going to be specialized are present in the template file as arguments of the monitor function. The presence of these arguments are necessary because of their use in the body of monitor function for calculating the table index and storing the value in the shared table. However, in FITTCHOOSER, none of the arguments are used in the function body. So whole arguments need to be copied.

In OFSPER, if the return type of the original function is of basic data types, the monitor function in the template file will also have the same return type. If it is of a `struct` data type, the monitor function in the template file use one of the basic data types in the template file. And, this pass will change it later.

The `LLVM` IR of the application program is used to get the missing arguments and return type of the functions contained in the template file. This `monitor` pass will do the following on the `LLVM` IR file.

— Changing return type and argument list of monitor function

1. Makes a copy of the original function.
2. Replace the body of the copied function with the body of monitor function.
3. Traverse through the arguments to find the arguments which are already present in the template file, and replace all its uses with the copied argument.
4. Find the return statement inside the function body and change the data type, if necessary.

— Changing the argument list and return type of other function declarations.

1. Create a copy of the original function for each declaration.
2. Replace the body of the copied function with the body of declaration.
3. Find each use of the above declaration.
4. Create a new call instruction with the exact arguments and return type.
5. Replace the use with the new instruction.

isPossible Pass

This pass is used to check the suitability of a function for applying the optimization. Even though this pass is required for both FITTCHOOSER and OFSPER, there is a slight difference in the suitability criteria for the two. So, there are two versions of this pass, one for FITTCHOOSER and another for OFSPER.

In OFSPER, the suitability of function is mainly depends on the arguments and their uses inside the function body. The current version of OFSPER is looking for the functions with at least one of the basic data type arguments is part of a loop. So, this pass will traverse through the argument list for finding such an argument. If it found one, it will output the details of the argument so that the tool can create the monitor function using this argument as one of the suitable candidate for specialization. Otherwise, the function is not considered for specialization.

In FITTCHOOSER, the suitability mainly depends on the return type. Since the monitor function is required to save the return value of the optimized versions, the monitor function should declare a variable with the exact data type. To reduce the complexity, the current version of FITTCHOOSER is looking for functions which are returning a value of basic data type.

optimize Pass

This pass is used by OFSPER for developing the specialized versions of a function. The working of this pass is very simple. The arguments on which the function is performing specialization and their values are provided to the pass as input. This pass will then traverse through the argument list to find the given argument for replacement. After finding the arguments, it will replace all their uses with the corresponding given value.

CHAPTER 5

CONCLUSION

Reducing the execution time of an application is one of the most discussed topic in the compiler community. Different optimization techniques, both static and dynamic, are presented in order to reduce the execution time of the application. Due to its nature, a static compiler has a limited visibility for optimization when it comes to taking into account the dynamic environment or behaviour of an application. Modern hardware features can boost the performance of an application, but software vendors are often limited to the lowest common denominator to maintain compatibility with the spectrum of processors used by their clients. In other hand, a dynamic optimizer have a better knowledge about run time environment and data. However, they do not have enough knowledge about the program in the binary code compared to the one present in the source code. To overcome this difficulty, some of the previous works relied on an intermediate representation of the program for applying optimization. In this thesis, we used intermediate representation produced by the LLVM compiler for gaining the information in the source code and performing optimizations on the process at run time.

This thesis presents two dynamic optimization techniques, FITTCHOOSER and OFSPER, both aim to improve the code performance during execution. They concentrate on the most used functions in the application to improve the execution time of the process. With the help of an extra function, known as *monitor function*, injected to the process at run time, they analyze and monitor the critical functions in the program. They produce different variants of the critical function based on the run time environment and inject and execute them instead of the original function in the process.

FITTCHOOSER replaces the critical functions in an application with another improved version of it. The main advantage of FITTCHOOSER is that it can compare the performance of multiple variations of the function dynamically to choose the best version. The

variants are produced by recompiling the intermediate representation of the function with different optimization flags. Since the recompilation is carried out on the running machine, the variants can benefit from the different hardware features provided by the machine, which are unknown during static compilation most of the time.

OFSPER tries to improve the critical functions by redirecting the function calls to better versions based on the values taken by arguments. It applies function specialization dynamically. Initially it monitors the values taken by the arguments of the critical functions for the repetition. If the values are repeating among different function calls, OFSPER creates a new version of the function exclusively for the repeating value. And it redirects the function calls to this new version whenever the function is called with the same value for the arguments.

5.1 Publications

1. HPCS

   Arif Ali Ap et al. « fittChooser: A Dynamic Feedback-Based Fittest Optimization Chooser ». In: *HPCS 2018 - 16th International Conference on High Performance Computing & Simulation - Special Session on Compiler Architecture, Design and Optimization*. Orléans, France, July 2018. URL: https://hal.inria.fr/hal-01808658

2. SAMOS

   Arif Ali Ap and Erven Rohou. « Dynamic Function Specialization ». In: *International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation*. Pythagorion, Samos, Greece, July 2017. URL: https://hal.inria.fr/hal-01597880

5.2 Further Extension

FITTCHOOSER shows a way to choose a set of parameters for an optimization technique which is fittest to the given run time environment. The value of such parameters may depend on both the machine hardware and the input data set of the application. However, OFSPER applied specialization without trying to find the above mentioned fittest parameter set. So it will be interesting to launch two levels of monitoring. First

one to find the value of the argument on which the specialization need to be applied as in OFSPER, and the second one to find the fittest optimization set for the specialized version as in FITTCHOOSER. Instead of creating one specialized version for one argument value, create different variations of it while applying specialization.

Currently both tools are trying to improve the performance of an application in terms of execution time. It will be interesting to focus on improvement not only in execution time but also in power conception.

Auto parallelization remains an open challenge. The main culprit here is the limited amount of static information which constrains tradition and research compilers. As an example, complex conditional statements and pointer aliasing which prevents parallelization. Runtime information can be used to resolve many of these conditionals and aliasing information which can help auto parallelization. Along with parallelization advanced optimization frameworks like the Polyhedral Model can be used to further optimize code

The current work focuses on improving binary performance, by function specialization, on similar architectures. However, the fundamental ideas behind current work can be used to improve application portability. For example consider a binary designed for x64 Xeon processors. Converting such an application for Intel KNL architecture is not just a matter of tuning parameters like vector length; it involves high level changes such as taking advantage of MCDRAM, generating codes aware of Numa architecture, etc. A more ambitious target would be to generate code for entirely different architectures such as GPUs. Existing frameworks can be leveraged to implement this objective. Even though current frameworks require source code, typical frameworks, such as MLIR, maintains an internal representation, which is used to automatically generate code for different architecture. A future extension of this work could convert binary such a high level representation and leverage existing frameworks for domain specific operations and code generation along with function specifications.

APPENDIX A

MONITOR FUNCTION FOR FITTCHOOSER

```
typedef struct {
    uint64_t count;
    int function;
    uint64_t avg_time_taken;
}shared_table;
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* The following 8 preprocessing directives are used to pass the
* function specific information to the program. Only these 8 lines
* are needed to rewrite for creating monitor function for another
* function. This part will be different for different functions.
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#define FUNCTION_NAME foo
#define SHARED_MEM_NAME "foo_padrone_shared_table"
#define RetTYPE    void
#define RETURN_DECLARATION
#define RETURN_ASSIGNMENT
#define RETURN_STATEMENT   return ;
#define TABLE_SIZE 30
#define FUNCTION_NAME_NEW(x) foo ##_## x
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#define SHM_SIZE TABLE_SIZE*sizeof(shared_table)

RetTYPE FUNCTION_NAME_NEW(original) (ARGUMENTS);

RetTYPE FUNCTION_NAME_NEW(0) (ARGUMENTS);
```

```
RetTYPE FUNCTION_NAME_NEW(1) (ARGUMENTS);
          • • •
RetTYPE FUNCTION_NAME_NEW(20) (ARGUMENTS);
#define QUANTA 20

RetTYPE FUNCTION_NAME_NEW(monitor) (ARGUMENTS){

    static int       number_of_versions  = 4;
    static long int flag_count_call = 0;
    unsigned int     table_index = TABLE_SIZE-1;
    unsigned int     tmp_index, tmp_argument;
    int              shmid;
    char             shared_mem_name[] = SHARED_MEM_NAME ;
    static           shared_table *table;

    uint64_t start, end, difference;
    RETURN_DECLARATION

    if(flag_count_call ==0){
        if((shmid = shm_open(shared_mem_name,  O_RDWR,S_IRUSR | S_IWUSR))< 0){
            perror("shm_open");
            exit(1);
        }
        if((table = mmap(NULL ,SHM_SIZE,PROT_READ | PROT_WRITE,MAP_SHARED ,
           shmid, 0))== NULL){
            perror("mmap");
            exit(1);
        }
    }

    flag_count_call ++;
    table_index     = (flag_count_call/QUANTA)% number_of_versions ;
    start = rdtsc();
    switch(table_index ) {
```

```
        case 0:
            RETURN_ASSIGNMENT FUNCTION_NAME_NEW(0) (CALL_ARGUMENTS);
            break;

        case 1:
            RETURN_ASSIGNMENT FUNCTION_NAME_NEW(1) (CALL_ARGUMENTS);
            break;
                    • • •
        case 20:
            RETURN_ASSIGNMENT FUNCTION_NAME_NEW(20) (CALL_ARGUMENTS);
            break;

        default:
            RETURN_ASSIGNMENT FUNCTION_NAME_NEW(original) (CALL_ARGUMENTS);
            break;
    }

    end = rdtsc();
    table[table_index].count++;
    table[table_index].avg_time_taken =
                    ( ( table[table_index].avg_time_taken *
                    ( table[table_index].count - 1) ) +
                    ( end - start ) )
                   / table[table_index].count ;

    RETURN_STATEMENT
}
```

APPENDIX B

MONITOR FUNCTION FOR OFSPER

```
typedef struct {
    int count;
    int function;
    union {
        int         int_argument;
        float       float_argument;
        double      double_argument;
        long int    long_int_argument;
        short int   short_int_argument;
    }args[6];
}shared_table;

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* The following 11 preprocessor directives are  used for passing information
* about the original function to the program. Only these 11 lines are
* needed to rewrite for creating monitor function for another function.
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#define FUNCTION_NAME SetupFastFullPelSearch
#define SHARED_MEM_NAME "SetupFastFullPelSearch_padrone_shared_table"
#define RetTYPE     void
#define TABLE_SIZE 5000
#define ORIGINAL_FUNCTION  20
#define ARGUMENTS   int ref, int list
#define CALL_ARGUMENTS   ref, list
#define INDEX_CALCULATING_ARGUMENT   ref<<(0*5) ^ list<<(1*5)
```

```c
#define ARGUMENT_CHECK     table[table_index].args[0].int_argument == ref && \
                         table[table_index].args[1].int_argument == list
#define ARGUMENT_SET      table[table_index].args[0].int_argument = ref; \
                         table[table_index].args[1].int_argument = list;
#define FUNCTION_NAME_NEW(x) SetupFastFullPelSearch ##_## x

/* * * * * * * * * * * * * * * * * * * * * * * * * * */

#define SHM_SIZE TABLE_SIZE*sizeof(shared_table)

RetTYPE FUNCTION_NAME_NEW(original) (ARGUMENTS);

RetTYPE FUNCTION_NAME_NEW(1) (ARGUMENTS);
RetTYPE FUNCTION_NAME_NEW(2) (ARGUMENTS);
             • • •
RetTYPE FUNCTION_NAME_NEW(20) (ARGUMENTS);

RetTYPE FUNCTION_NAME_NEW(monitor) (ARGUMENTS){
    static int flag_first_call = 0;
    unsigned int table_index = TABLE_SIZE-1;
    unsigned int  tmp_index, tmp_argument;
    int shmid;
    char shared_mem_name[] = SHARED_MEM_NAME ;
    static shared_table *table;
    if(flag_first_call ==0){
        if((shmid = shm_open(shared_mem_name,  O_RDWR,S_IRUSR | S_IWUSR))< 0){
            perror("shm_open");
            exit(1);
        }
        if((table = mmap(NULL,SHM_SIZE,PROT_READ | PROT_WRITE,MAP_SHARED ,
             shmid, 0))== NULL){
            perror("mmap");
            exit(1);
        }
```

```
        flag_first_call = 1;
    }
    table_index      = (unsigned int ) INDEX_CALCULATING_ARGUMENT;
    table_index      = table_index ^ (table_index>>16);
    table_index      = table_index & 0x00FFF;

    if(ARGUMENT_CHECK){
        table[table_index].count++;
    }
    else if(table[table_index].count == 0){
        ARGUMENT_SET;
        table[table_index].count++;
    }
    else{
        table_index = TABLE_SIZE -1;
        table[table_index].count++;
    }

    switch(table[table_index].function){
        case 1:
            return FUNCTION_NAME_NEW(1) (CALL_ARGUMENTS);

        case 2:
            return FUNCTION_NAME_NEW(2) (CALL_ARGUMENTS);

                        • • •
        case 20:
            return FUNCTION_NAME_NEW(20) (CALL_ARGUMENTS);

        default:
            return FUNCTION_NAME_NEW(original) (CALL_ARGUMENTS);
    }
}
```

Bibliography

[Ap+18] Arif Ali Ap et al. « fittChooser: A Dynamic Feedback-Based Fittest Optimization Chooser ». In: *HPCS 2018 - 16th International Conference on High Performance Computing & Simulation - Special Session on Compiler Architecture, Design and Optimization*. Orléans, France, July 2018. URL: https://hal.inria.fr/hal-01808658.

[AR17] Arif Ali Ap and Erven Rohou. « Dynamic Function Specialization ». In: *International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation*. Pythagorion, Samos, Greece, July 2017. URL: https://hal.inria.fr/hal-01597880.

[Ash+18] Amir H Ashouri et al. « A survey on compiler autotuning using machine learning ». In: *arXiv preprint arXiv:1801.04405* (2018).

[Asl+01] Vishal Aslot et al. « SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance ». In: *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2001 West Lafayette, IN, USA, July 30–31, 2001 Proceedings*. Ed. by Rudolf Eigenmann and Michael J. Voss. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–10. ISBN: 978-3-540-44587-6. DOI: 10.1007/3-540-44587-0_1. URL: http://dx.doi.org/10.1007/3-540-44587-0_1.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. ISBN: 0-201-10088-6. URL: http://www.worldcat.org/oclc/12285707.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. « Compiler Transformations for High-performance Computing ». In: *ACM Comput. Surv.* 26.4 (Dec. 1994), pp. 345–420. ISSN: 0360-0300. DOI: 10.1145/197405.197406. URL: http://doi.acm.org/10.1145/197405.197406.

[Bru04] Derek Bruening. « Efficient, Transparent, and Comprehensive Runtime Code Manipulation ». PhD thesis. MIT, Sept. 2004.

[BZA12] Derek Bruening, Qin Zhao, and Saman Amarasinghe. « Transparent Dynamic Instrumentation ». In: *SIGPLAN Not.* 47.7 (Mar. 2012), pp. 133–144. ISSN: 0362-1340. DOI: 10.1145/2365864.2151043. URL: http://doi.acm.org/10.1145/2365864.2151043.

[CCD17] João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. « Chapter 5 - Source code transformations and optimizations ». In: *Embedded Computing for High Performance*. Ed. by João M.P. Cardoso, José Gabriel F. Coutinho, and Pedro C. Diniz. Boston: Morgan Kaufmann, 2017, pp. 137–183. ISBN: 978-0-12-804189-5. DOI: https://doi.org/10.1016/B978-0-12-804189-5.00005-3. URL: http://www.sciencedirect.com/science/article/pii/B9780128041895000053.

[CDS03] B. Childers, J. W. Davidson, and M. L. Soffa. « Continuous compilation: a new approach to aggressive and adaptive code transformation ». In: *Proceedings International Parallel and Distributed Processing Symposium*. Apr. 2003, 10 pp.-. DOI: 10.1109/IPDPS.2003.1213375.

[Che+10] Dehao Chen et al. « Taming Hardware Event Samples for FDO Compilation ». In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10. Toronto, Ontario, Canada: ACM, 2010, pp. 42–52. ISBN: 978-1-60558-635-9. DOI: 10.1145/1772954.1772963. URL: http://doi.acm.org/10.1145/1772954.1772963.

[Cli90] William D. Clinger. « How to Read Floating Point Numbers Accurately ». In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. PLDI '90. White Plains, New York, USA: ACM, 1990, pp. 92–101. ISBN: 0-89791-364-7. DOI: 10.1145/93542.93557. URL: http://doi.acm.org/10.1145/93542.93557.

[CLM04] Charles Consel, Julia L. Lawall, and Anne-Françoise Le Meur. « A tour of Tempo: a program specializer for the C language ». In: *Science of Computer Programming* 52.1–3 (2004). Special Issue on Program Transformation. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2004.03.011. URL: http://www.sciencedirect.com/science/article/pii/S0167642304000553.

[DH79] J. J. Dongarra and A. R. Hinds. « Unrolling loops in fortran ». In: *Software: Practice and Experience* 9.3 (1979), pp. 219–226. ISSN: 1097-024X. DOI: 10.1002/spe.4380090307. URL: http://dx.doi.org/10.1002/spe.4380090307.

[DR] Artem Dinaburg and Andrew Ruef. *McSema: Static translation of x86 instructions to LLVM*. https://blog.trailofbits.com/2014/06/23/a-preview-of-mcsema/. Accessed: 2016-11-02.

[EAH97] Kemal Ebcioglu, Erik Altman, and Erdem Hokenek. « A JAVA ILP machine based on fast dynamic compilation ». In: *IN IEEE MASCOTS INTERNATIONAL WORKSHOP ON SECURITY AND E CIENCY ASPECTS OF JAVA*. 1997.

[Eke16] Per Ekemark. *Static Multi-Versioning for Efficient Prefetching*. 2016.

[Fis81] J. A. Fisher. « Trace Scheduling: A Technique for Global Microcode Compaction ». In: *IEEE Trans. Comput.* 30.7 (July 1981), pp. 478–490. ISSN: 0018-9340. DOI: 10.1109/TC.1981.1675827. URL: http://dx.doi.org/10.1109/TC.1981.1675827.

[GSK01] G. Goumas, A. Sotiropoulos, and N. Koziris. « Minimizing completion time for loop tiling with computation and communication overlapping ». In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*. Apr. 2001, 10 pp.-. DOI: 10.1109/IPDPS.2001.924976.

[Hal+15] Nabil Hallou et al. « Dynamic Re-Vectorization of Binary Code ». In: *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation - SAMOS XV*. Agios Konstantinos, Greece, July 2015. URL: https://hal.inria.fr/hal-01155207.

[Haw+15] Byron Hawkins et al. « Optimizing Binary Translation of Dynamically Generated Code ». In: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '15. San Francisco, California: IEEE Computer Society, 2015, pp. 68–78. ISBN: 978-1-4799-8161-8. URL: http://dl.acm.org/citation.cfm?id=2738600.2738610.

[HC89] W. W. Hwu and P. P. Chang. « Achieving High Instruction Cache Performance with an Optimizing Compiler ». In: *SIGARCH Comput. Archit. News* 17.3 (Apr. 1989), pp. 242–251. ISSN: 0163-5964. DOI: 10.1145/74926.74953. URL: http://doi.acm.org/10.1145/74926.74953.

[HDT16] Byron Hawkins, Brian Demsky, and Michael B. Taylor. « BlackBox: Lightweight Security Monitoring for COTS Binaries ». In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO '16. Barcelona, Spain: ACM, 2016, pp. 261–272. ISBN: 978-1-4503-3778-6. DOI: 10.1145/2854038.2854062. URL: http://doi.acm.org/10.1145/2854038.2854062.

[Hen06] John L. Henning. « SPEC CPU2006 Benchmark Descriptions ». In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 0163-5964. DOI: 10.1145/1186736.1186737. URL: http://doi.acm.org/10.1145/1186736.1186737.

[Hon+12] Ding-Yong Hong et al. « HQEMU: A Multi-threaded and Retargetable Dynamic Binary Translator on Multicores ». In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO '12. San Jose, California: ACM, 2012, pp. 104–113. ISBN: 978-1-4503-1206-6. DOI: 10.1145/2259016.2259030. URL: http://doi.acm.org/10.1145/2259016.2259030.

[HRC16] Nabil Hallou, Erven Rohou, and Philippe Clauss. « Runtime Vectorization Transformations of Binary Code ». In: *International Journal of Parallel Programming* (2016), pp. 1–30. ISSN: 1573-7640. DOI: 10.1007/s10766-016-0480-z. URL: http://dx.doi.org/10.1007/s10766-016-0480-z.

[HS02] Edin Hodzic and Weijia Shang. « On Time Optimal Supernode Shape ». In: *IEEE Trans. Parallel Distrib. Syst.* 13.12 (Dec. 2002), pp. 1220–1233. ISSN: 1045-9219. DOI: 10.1109/TPDS.2002.1158261. URL: https://doi.org/10.1109/TPDS.2002.1158261.

[Kis+00] T. Kisuki et al. *Iterative Compilation in Program Optimization*. 2000.

[KKO00] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. « Combined selection of tile sizes and unroll factors using iterative compilation ». In: *Proceedings 2000 International Conference on Parallel Architectures and Compilation*

Techniques (Cat. No.PR00622). Oct. 2000, pp. 237–246. DOI: 10.1109/PACT.2000.888348.

[KKO02] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. « Iterative Compilation ». In: *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation — SAMOS*. Ed. by Ed F. Deprettere, Jürgen Teich, and Stamatis Vassiliadis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 171–187. ISBN: 978-3-540-45874-6. DOI: 10.1007/3-540-45874-3_10. URL: https://doi.org/10.1007/3-540-45874-3_10.

[KM94] Ken Kennedy and Kathryn S. McKinley. « Maximizing loop parallelism and improving data locality via loop fusion and distribution ». In: *Languages and Compilers for Parallel Computing*. Ed. by Utpal Banerjee et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 301–320. ISBN: 978-3-540-48308-3.

[LA04] Chris Lattner and Vikram Adve. « LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation ». In: *International Symposium on Code Generation and Optimization (CGO'04)*. 2004.

[Leh16] Jan-Patrick Lehr. « Counting performance: hardware performance counter and compiler instrumentation ». In: *Informatik 2016, 46. Jahrestagung der Gesellschaft für Informatik, 26.-30. September 2016, Klagenfurt, Österreich*. Ed. by Heinrich C. Mayr and Martin Pinzger. Vol. P-259. LNI. GI, 2016, pp. 2187–2198. ISBN: 978-3-88579-653-4. URL: http://subs.emis.de/LNI/Proceedings/Proceedings259/article126.html.

[Llv16] *Writing an LLVM Pass*. URL: http://llvm.org/docs/WritingAnLLVMPass.html (visited on 09/26/2016).

[Luk+05] Chi-Keung Luk et al. « Pin: building customized program analysis tools with dynamic instrumentation ». In: *pldi*. Chicago, IL, USA, 2005.

[Mac+17] R. S. Machado et al. « Comparing Performance of C Compilers Optimizations on Different Multicore Architectures ». In: *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. Oct. 2017, pp. 25–30. DOI: 10.1109/SBAC-PADW.2017.13.

[Mic+07] Pierre Michaud et al. « A study of thread migration in temperature-constrained multicores ». In: *j-TACO* 4.2 (June 2007). ISSN: 1544-3566 (print), 1544-3973 (electronic). DOI: http://doi.acm.org/10.1145/1250727.1250729.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN: 1-55860-320-4.

[Net04] Nicholas Nethercote. *Dynamic binary analysis and instrumentation*. Tech. rep. UCAM-CL-TR-606. University of Cambridge, Computer Laboratory, Nov. 2004. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf.

[NS07] Nicholas Nethercote and Julian Seward. « Valgrind: a framework for heavy-weight dynamic binary instrumentation ». In: *PLDI*. San Diego, California, USA, 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746. URL: http://doi.acm.org/10.1145/1250734.1250746.

[Nuz+13] Dorit Nuzman et al. « JIT Technology with C/C++: Feedback-directed Dynamic Recompilation for Statically Compiled Languages ». In: *ACM Trans. Archit. Code Optim.* 10.4 (Dec. 2013), 59:1–59:25. ISSN: 1544-3566. DOI: 10.1145/2541228.2555315. URL: http://doi.acm.org/10.1145/2541228.2555315.

[Pén+16] Pierre-Yves Péneau et al. « Loop Optimization in Presence of STT-MRAM Caches: a Study of Performance-Energy Tradeoffs ». In: *PATMOS: Power and Timing Modeling, Optimization and Simulation*. Proceedings of the 26th International Workshop on Power and Timing Modeling, Optimization and Simulation. Bremen, Germany, Sept. 2016, pp. 162–169. DOI: 10.1109/PATMOS.2016.7833682. URL: https://hal.inria.fr/hal-01347354.

[PH90] Karl Pettis and Robert C. Hansen. « Profile Guided Code Positioning ». In: *SIGPLAN Not.* 25.6 (June 1990), pp. 16–27. ISSN: 0362-1340. DOI: 10.1145/93548.93550. URL: http://doi.acm.org/10.1145/93548.93550.

[POL] *PolyBench/C v4.1: the Polyhedral Benchmark suite*. URL: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. « The Java HotSpot™ Server Compiler ». In: *Proc. of the Java Virtual Machine Research and Technology Symposium*. Monterey, CA, USA, Apr. 2001.

[Rio+14] Emmanuel Riou et al. « PADRONE: a Platform for Online Profiling, Analysis, and Optimization ». In: *DCE 2014 - International workshop on Dynamic Compilation Everywhere*. Vienne, Austria, Jan. 2014. URL: https://hal.inria.fr/hal-00917950.

[SA05] Mark Stephenson and Saman Amarasinghe. « Predicting Unroll Factors Using Supervised Classification ». In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 123–134. ISBN: 0-7695-2298-X. DOI: 10.1109/CGO.2005.29. URL: http://dx.doi.org/10.1109/CGO.2005.29.

[San+13] Henrique Nazare Santos et al. « Just-in-time Value Specialization ». In: *International Symposium on Code Generation and Optimization (CGO)*. CGO '13. USA: IEEE Computer Society, 2013, pp. 1–11. ISBN: 978-1-4673-5524-7. DOI: 10.1109/CGO.2013.6495006. URL: http://dx.doi.org/10.1109/CGO.2013.6495006.

[SC15] Aravind Sukumaran-Rajam and Philippe Clauss. « The Polyhedral Model of Nonlinear Loops ». In: *ACM Trans. Archit. Code Optim.* 12.4 (Dec. 2015), 48:1–48:27. ISSN: 1544-3566. DOI: 10.1145/2838734. URL: http://doi.acm.org/10.1145/2838734.

[SG06a] Florian Schneider and Thomas R. Gross. « Using Platform-Specific Performance Counters for Dynamic Compilation ». In: *Languages and Compilers for Parallel Computing: 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers*. Ed. by Eduard Ayguadé et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 334–346. ISBN: 978-3-540-69330-7. DOI: 10.1007/978-3-540-69330-7_23. URL: https://doi.org/10.1007/978-3-540-69330-7_23.

[SG06b] Florian Schneider and Thomas R. Gross. « Using Platform-Specific Performance Counters for Dynamic Compilation ». In: *Languages and Compilers for Parallel Computing*. Ed. by Eduard Ayguadé et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 334–346.

[Smi00] Michael D. Smith. « Overcoming the Challenges to Feedback-directed Optimization (Keynote Talk) ». In: *SIGPLAN Not.* 35.7 (Jan. 2000), pp. 1–11. ISSN: 0362-1340. DOI: 10.1145/351403.351408. URL: http://doi.acm.org/10.1145/351403.351408.

[SPE] SPEC.org. *SPEC CPU2017*. https://www.spec.org/cpu2017.

[SRS17] Arjun Suresh, Erven Rohou, and André Seznec. « Compile-time Function Memoization ». In: *Proceedings of the 26th International Conference on Compiler Construction*. CC 2017. Austin, TX, USA: ACM, 2017, pp. 45–54. ISBN: 978-1-4503-5233-8. DOI: 10.1145/3033019.3033024. URL: http://doi.acm.org/10.1145/3033019.3033024.

[Suk+14] Aravind Sukumaran-Rajam et al. « Speculative Program Parallelization with Scalable and Decentralized Runtime Verification ». In: *Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Cham: Springer International Publishing, 2014, pp. 124–139. ISBN: 978-3-319-11164-3.

[Sur+15] Arjun Suresh et al. « Intercepting Functions for Memoization: A Case Study Using Transcendental Functions ». In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.2 (July 2015), p. 23. DOI: 10.1145/2751559. URL: https://hal.inria.fr/hal-01178085.

[SW90] Guy L. Steele and Jon L. White. « How to Print Floating-Point Numbers Accurately ». In: *PLDI*. 1990.

[SZW06] Saravanan Sinnadurai, Qin Zhao, and Weng-Fai Wong. *Transparent Runtime Shadow Stack: Protection against malicious return address modifications*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.5702&rep=rep1&type=pdf. 2006.

[Wat+17] Neftali Watkinson et al. « Using Hardware Counters to Predict Vectorization ». In: (2017). URL: https://pdfs.semanticscholar.org/e42b/f4c20e8236093b24ebca701d1c38d7bd5591.pdf.

[Wha99] John Whaley. *Dynamic Optimization through the use of Automatic Runtime Specialization*. 1999.

[Wic+14] Baptiste Wicht et al. « Hardware Counted Profile-Guided Optimization ». In: *CoRR* abs/1411.6361 (2014). arXiv: 1411.6361. URL: http://arxiv.org/abs/1411.6361.

[Zho+14] Mingzhou Zhou et al. « Space-efficient Multi-versioning for Input-adaptive Feedback-driven Program Optimizations ». In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 2014, pp. 763–776. ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660229. URL: http://doi.acm.org/10.1145/2660193.2660229.

LIST OF FIGURES

1 Progression de la passe d'optimisation. 10
2 Séquence d'appel: Normal vs Spécialisation 13

1.1 Iterative Compilation Flow Chart given in [KKO02]. 29

2.1 Progression of the Optimization Pass. 40
2.2 The shared Monitoring table is accessed by both the application and FITTCHOOSER. 45
2.3 Overhead of FITTCHOOSER. 48
2.4 Overall speedup under FITTCHOOSER. 49
2.5 Performance of critical function variations. 51

3.1 Call sequence: Normal vs Specialization 60
3.2 OFSPER attached to the application process 64
3.3 The shared monitoring table is accessed by both the application and OFSPER 65
3.4 Impact of specialization on execution time of a function (lower is better) 69
3.5 Overhead by profiling and monitoring 71
3.6 Speedups 72

LIST OF TABLES

1.1 Constant Propagation 22

2.1 Example of the shared Monitoring table. 44
2.2 Standard Deviation 50

3.1 Repeatability of arguments 58
3.2 Monitoring Table 61
3.3 Modified Monitoring Table 62
3.4 Monitoring table for the pure function `exp2` 62

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Transformation binaire de niveau de fonction dynamique axée sur les performances

Mot clés : Compilateurs, optimisation dynamique, remplacement de fonctions en ligne

Résumé : Les évolutions de l'architecture des processeurs visent à améliorer les performances des applications, mais les éditeurs de logiciels sont souvent limités au plus petit dénominateur commun afin de maintenir la compatibilité avec la diversité du matériel de leurs clients. Avec des informations plus détaillées, un compilateur peut générer un code plus efficace. Même si le modèle de processeur est connu, les fabricants ne divulguent pas de nombreux détails pour des raisons de confidentialité. En outre, l'efficacité de nombreuses techniques d'optimisation peut varier en fonction des entrées du programme.Cette thèse introduit deux outils, FITTCHOOSER et OFSPER, qui effectuent des optimisations au niveau des fonctions les mieux adaptées à l'environnement d'exécution et aux données en cours. FITTCHOOSER explore de manière dynamique les spécialisations des fonctions les plus gourmandes en ressources d'un programme pour choisir la version la plus adaptée – non seulement à l'environnement d'exécution en cours, mais également à l'exécution en cours du programme. OFSPER applique une *spécialisation de fonction dynamique*, c'est-à-dire la spécialisation de fonctions dans une application sur un processus en cours d'exécution. Cette technique capture les valeurs réelles des arguments lors de l'exécution du programme et, si rentables, crée des versions spécialisées et les inclut au moment de l'exécution.

Title: Performance Centric Dynamic Function Level Binary Transformation

Keywords: Compilers, Dynamic Optimization, Online Function Replacement

Abstract: Modern hardware features can boost the performance of an application, but software vendors are often limited to the lowest common denominator to maintain compatibility with the spectrum of processors used by their clients. Given more detailed information about the hardware features, a compiler can generate more efficient code, but even if the exact CPU model is known, manufacturer confidentiality policies leave substantial uncertainty about precise performance characteristics. In addition, the effectiveness of many optimization techniques can vary depending on the inputs to the program. This thesis introduces two tools, FITTCHOOSER and OFSPER, to do function-level optimizations most suitable for the current runtime environment and data. FITTCHOOSER dynamically explores specializations of a program's most processor-intensive functions to choose the fittest version—not just specific to the current runtime environment, but also specific to the current execution of the program. OFSPER applies *dynamic function specialization*, applying function specialization on a running process, to an application. This technique captures the actual values of arguments during execution of the program and, when profitable, creates specialized versions and include them at runtime.