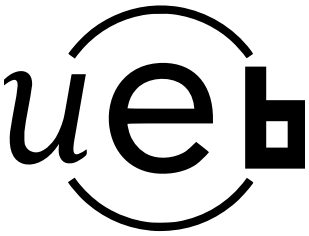
Extraction and traceability of annotations for WCET estimation

Hanbing Li

N° d'ordre : 00000 ANNÉE 2015



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Hanbing LI

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Aléatoires
Composante Universitaire (Université Rennes 1)

Extraction and Traceability of Annotations for WCET Estimation

Thèse soutenue à Rennes
le 9 Octobre 2015

devant le jury composé de :

CHRISTINE ROCHANGE
Professeur à l'Université Toulouse 3 Paul Sabatier / *Rapporteur*

PHILIPPE CLAUSS
Professeur à l'Université de Strasbourg / *Rapporteur*

FLORIAN BRANDNER
Maître de conférence à ENSTA ParisTech / *Examinateur*

STEVEN DERRIEN
Professeur à l'Université de Rennes 1 / *Examinateur*

ISABELLE PUAUT
Professeur à l'Université de Rennes 1 / *Co-Directeur de thèse*

ERVEN ROHOU
Directeur de Recherche INRIA / *Co-directeur de thèse*

To see a world in a grain of sand
And heaven in a wild flower
Hold infinity in the palms of your hand
And eternity in an hour.
—William Blake

RESUMÉ EN FRANÇAIS

Extraction et traçabilité d'annotations pour l'estimation de WCET

Motivation et techniques d'estimation de WCET

Dans les systèmes temps-réel dur, il est nécessaire de connaître le pire temps d'exécution (en anglais Worst Case Execution Time [WCET]), de portions de code, pour démontrer que le système respecte ses contraintes de temps, dans toutes les situations, y compris la pire. Les estimations du WCET doivent être sûres et le plus précises possibles. La sûreté signifie que l'estimation doit être supérieure ou égale à toute durée réelle d'exécution. La précision rend l'estimation utile, elle permet d'éviter de surestimer le besoin en ressource processeur : le WCET estimé doit être le plus proche possible du WCET réel.

Les techniques d'estimation de WCET peuvent être classées en deux catégories : les méthodes statiques et celles basées sur des mesures. Les méthodes statiques assurent la sûreté, car elles surestiment le WCET. L'estimation du WCET avec ces méthodes est calculée au niveau du code machine, parce que la durée des opérations élémentaires (instructions) ne peut pas être obtenue à un niveau plus élevé.

La méthode d'estimation statique de WCET utilisée dans cette thèse est la méthode IPET (énumération implicite des chemins ou Implicit Path Enumeration Technique). Cette méthode opère sur le graphe de flot de contrôle (CFG pour Control Flow Graph), extrait à partir du code binaire. IPET modèle le problème de calcul de WCET comme un problème de programmation linéaire en nombres entiers (PLNE).

Les information de flot sur les programme (bornes de boucles, chemins infaisables etc.) sont nécessaires pour calculer des WCET précis. Les information de flot peuvent être obtenues en utilisant des techniques d'analyse statique, ou ajoutées manuellement par par le développeur d'applications par le biais d'annotations. Dans les deux situations, il est pratique d'extraire ou d'exprimer des informations de flot au niveau du code source.

Les compilateurs modernes appliquent des centaines d'optimisations pour améliorer les performances des programmes. Certaines d'entre elles modifient radicalement le flot de contrôle du programme, rendant difficile la correspondance entre la structure du code binaire et le code source original. Ainsi, les annotations au niveau du code source ne peuvent pas être utilisées directement.

Pour résoudre ce problème, nous proposons une infrastructure logicielle de transformation des annotations, du code source au code binaire.

L'infrastructure de transformation

Notre infrastructure de transformation transmet les informations de flot à partir du code source dans notre cas, C ou tout autre langage compile vers le code machine. Les transformations sont exprimées de manière d'abstraite, indépendamment de l'infrastructure de

compilation dans lequel elles seront intégrées.

L'infrastructure de transformation, pour chaque optimisation du compilateur, définit un ensemble de formules, qui réécrivent les contraintes de flot disponibles en de nouvelles contraintes. Elle supporte n'importe quelle contrainte linéaire sur les nombres d'exécutions des blocs de base. Les bornes de boucle et les chemins infaisables, ainsi que toutes les autres informations de flot seront toutes exprimées au final comme des contraintes linéaires.

Il existe trois règles de réécriture de base pour transformer les informations de flot : *la règle de changement*, *la règle de suppression* et *la règle d'addition*.

La règle de changement

Cette règle est utilisée pour exprimer les changements de nombres d'exécutions des blocs de base, ainsi que les changements des bornes de boucles, résultant d'optimisations de compilation. Elle est exprimée par $\alpha \rightarrow \beta$, ce qui signifie que α est substitué par β dans les contraintes.

Cette règle contient deux cas :

Le premier cas est le changement dans le nombre d'exécutions d'un bloc de base. Dans ce cas, α est f_i, où i est un des blocs de base dans le CFG avant optimisation. β est une expression $\{C_j + \sum_{j \in new\_CFG} M_j \times f_j\}$, où C est une constante et M est un coefficient multiplicatif, qui peut être soit une constante entière non-négative, soit un intervalle [a,b], soit un intervalle [a,+∞) où a et b sont des constantes non-négatives.

Le second cas est le changement dans les bornes d'une boucle. α est alors une contrainte de la borne de boucle $L_x \langle l_{bound}, u_{bound} \rangle$, où $L_x \subset CFG\_original$, et β est $L_y \langle l_{bound'}, u_{bound'} \rangle$. Les nouvelles bornes inférieures et supérieures de boucle $l_{bound'}$ and $u_{bound'}$ peuvent être des constantes entières non négatives ou toute expression impliquant uniquement des constantes (e.g., valeur plafond ou plancher d'une fraction) dont le résultat est un nombre entier non-négatif.

La règle de suppression

Cette règle est utilisée à chaque fois qu'un bloc de base ou une boucle est supprimée du CFG en raison d'une optimisation. Nous l'exprimons comme $\alpha \rightarrow \emptyset$. α peut être f_i ($i \in CFG\_original$) ou $L_x \langle l_{bound}, u_{bound} \rangle$ ($L_x \subset CFG\_original$) en fonction de l'objet (bloc de base, boucle) qui est supprimé. Grâce à cette transformation, α est supprimé des contraintes.

La règle d'addition

Cette dernière règle est destinée à être utilisée par des optimisations qui ajoutent des nouveaux objets (bloc de base/boucle) dans le CFG. Quand un nouveau terme est introduit dans le CFG, la nouvelle contrainte est ajoutée directement. La contrainte doit être linéaire, et seulement impliquer des objets (blocs de base, boucles) du nouveau CFG.

Après l'application des règles de transformation, nous devons normaliser les nouvelles contraintes, et introduisons pour ce faire des règles de normalisation.

Notre infrastructure de transformation prend en charge les optimisations du compilateur les plus courantes. Pour la grande majorité des optimisations, le nouveau WCET estimé est

meilleur (plus faible) que l'original.

Mise en œuvre dans l'infrastructure de compilateur LLVM

Nous avons intégré les règles de transformation dans l'infrastructure de compilation LLVM, version 3.4. LLVM comporte trois phases. La première phase est le frontal du compilateur, nommé *clang*, qui analyse, valide et diagnostique les erreurs dans le code C/C++. Il traduit ensuite le code vers la Representation Intermédiaire (IR) de LLVM. Dans la seconde phase, nommée *opt* (l'optimiseur de LLVM), une série d'analyses et optimisations est effectuée, avec comme objectif l'amélioration de la qualité du code. Enfin, la dernière étape du compilateur, nommé *codegen* produit du code machine natif à partir de la représentation intermédiaire.

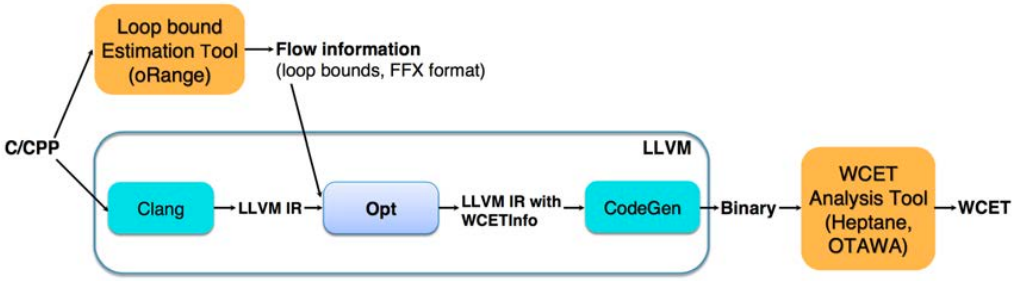


Figure 1 – La mise en œuvre de la traçabilité dans LLVM

Nous avons ajouté un nouveau type d'information à LLVM, nommé *WCETInfo*, que nous attachons au programme. Son objectif est d'associer à chaque boucle (objet "loop" dans LLVM) l'estimation correspondante de ses bornes inférieures et supérieures. Les optimisations disponibles dans LLVM peuvent préserver, mettre à jour ou supprimer les informations WCETInfo.

Comme le montre la figure 1, l'information sur les bornes de boucles est tout d'abord extraite par un outil d'estimation de borne de boucle et est stockée dans un fichier conforme au format FFX (format d'annotation basé sur XML). Notre version modifiée de LLVM lit les informations de flot à partir du fichier FFX et de les stocke dans WCETInfo. Lors de la compilation, les informations WCETInfo sont prises en compte au sein de nos optimisations LLVM modifiées. Enfin, la générateur de code modifié ajoute les bornes de boucles finales dans le code binaire (dans une section spécifique du fichier binaire), pour une utilisation ultérieure dans le calcul du WCET.

Résultats expérimentaux

Pour les optimisations sans vectorisation, nous examinons d'abord l'impact sur le WCET estimé de *toutes les* optimisations du niveau `-O3` avec l'outil d'analyse statique de WCET Heptane[1]. Ensuite, nous évaluons l'impact de chaque optimisation en la désactivant, et comparont son *effet négatif* (nous avons d'abord désactivé une optimisation sur n, puis deux sur n).

[1] https://team.inria.fr/alf/software/heptane

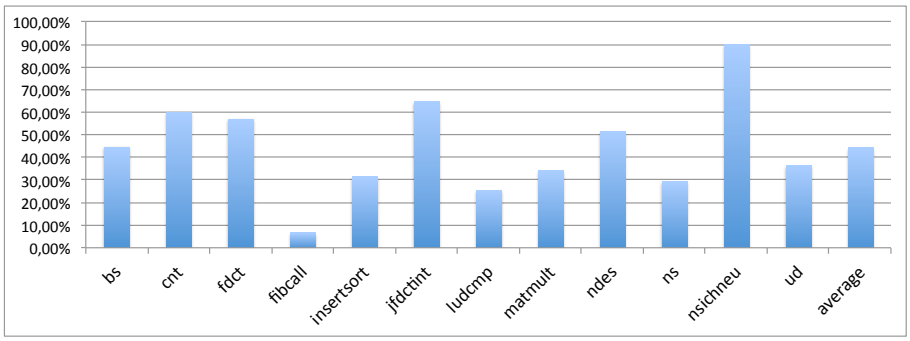


Figure 2 – Impact des optimisations (`-O3`) sur WCET. L'axe du Y représente le WCET avec des optimisations, normalisée par rapport à WCET sans optimisation (`-O0`)

La figure 2 montre l'impact des optimisations du compilateur sur le WCET estimé par Heptane. La figure montre que nous pouvons obtenir cette estimation grâce à notre infrastructure logicielle de transformation. Nous sommes par conséquent capables de transformer toutes les informations de flot à partir du code C vers le code binaire sans perte d'information. Nous observons également que l'option `-O3` permet une réduction importante du WCET estimé.

Pour les optimisations de vectorisation, parce que les outils d'estimation de WCET auxquels nous avons accès ne supportent actuellement pas les instructions SIMD (Single Instruction Multiple Data), nous effectuons des mesures sur du matériel réel pour collecter les temps d'exécution réel pour des codes à chemin unique. Grâce aux expériences sur les architectures Intel x86 et ARMv7 avec les ensembles de benchmarks TSVC et gcc-loops, nous pouvons faire des observations similaires et concluons que la vectorisation réduit les WCETs, et qu'elle est plus efficace sur l'architecture Intel étudiée.

Conclusion

Les concepteurs de systèmes temps-réel ont besoin de calculer les WCET des composants de leurs systèmes. Ceci est accompli en combinant des annotations prévues au niveau du code source par le programmeur (par exemple les bornes de boucle) et générées au niveau du code machine par le compilateur. Cette combinaison est possible si une équivalence est maintenue entre les deux représentations. Les optimisations du compilateur brisent généralement cette équivalence. Nous proposons par conséquent une infrastructure logicielle, construite dans le compilateur LLVM, qui propage les informations pour toutes les optimisations du compilateur. Nous illustrons notre mécanisme sur les bornes de boucles, et nous montrons que de nombreuses optimisations peuvent être activées.

Notre travail en cours concernant la traçabilité de C à binaire vise à étendre la traçabilité des informations au delà bornes de boucles (par exemple en considérant les branchements mutuellement exclusifs). Les autres travaux comprennent l'introduction d'informations contextuelles, des bornes globales de boucles, le format de sortie et un support supplémentaire pour les jeux d'instructions vertoriels.

Remerciements

Je tiens tout d'abord à remercier mes directeurs de thèse, Erven Rohou et Isabelle Puaut pour l'aide compétente qu'ils m'ont apportée, pour leur patience et leur encouragement à finir mon travail.

J'exprime tous mes remerciements à l'ensemble des membres de mon jury de thèse: Mesdames Christine Rochange et Isabelle Puaut et Messieurs Philippe Clauss, Florian Brandner, Steven Derrien et Erven Rohou.

Je remercie mes parents Tianlin LI et Shubo LENG et ma femme Kun He qui me soutiennent dans ma thèse et ma vie. Je vous aime.

Je remercie tous les membres et ex-membres de l'équipe ALF pour le climat sympathique dans lequel ils m'ont permis de travailler.

J'adresse toute ma gratitude à tous les membres du projet W-SEPT.

Je remercie tous mes ami(e)s avec lesquels j'ai partagé tous ces moments de doute et de plaisir.

Publications

- Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation [LPR14]. Hanbing Li, Isabelle Puaut, Erven Rohou - 22nd International Conference on Real-Time Networks and Systems RTNS 2014, October 8-10, 2014, Versailles, France.
- Tracing Flow Information for Tighter WCET Estimation: Application to Vectorization [LPR15]. Hanbing Li, Isabelle Puaut, Erven Rohou - 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications RTCSA 2015, August 19-21, 2015, Hong Kong, China.

Contents

Introduction

Motivation of the thesis

As a thesis of computer science, let us start it from "computer". A computer is a general-purpose device that can be programmed to carry out a set of arithmetic or logical operations automatically. In 1941, Z3, the world's first working electromechanical programmable, fully automatic digital computer was built by Zuse. Then ABC (Atanasoff-Berry Computer), the first "automatic electronic digital computer" was developed by John Vincent Atanasoff and Clifford E. Berry of Iowa State University in 1942. In 1946, ENIAC (Electronic Numerical Integrator and Computer), the first electronic programmable computer and the first Turing-complete device was built and announced in the US. Then the computer technologies began to develop and expand rapidly, and thanks to all the computer scientists and their works, computer systems become an essential part of human life and make our lives convenient, wonderful and magnificent.

At the beginning, personal computers, servers and supercomputers were the focus for concern. However, with the continued miniaturization of computing resources, and advancements in portable battery life, portable computer systems grew in popularity. The remarkable example is mobile phone or smartphone. These computer systems belong to embedded systems.

In contrast with personal computers, embedded systems are systems with a dedicated function within a larger mechanical or electrical system. The processor and software in an embedded system is usually unnoticed by the users.

Nowadays, embedded systems control many devices in our daily life. Besides, the usage and complexity varies wildly, *e.g.* microwave oven, digital watch, MP3 player, etc. A complex example is modern cars, which contain many embedded systems: anti-lock braking system (ABS), vehicle monitoring system, car entertainment system and so on.

An embedded system is called "real-time" when it is designed in order to guarantee that real-time application requests will be served within prespecified timing constraints. ABS in the cars is a remarkable example of real-time system.

In real-time systems, knowing the Worst-Case Execution Time (WCET) of pieces of software is required to demonstrate that the system meets its timing constraints for a given hardware platform (with different inputs, in a given hardware and operating system), including the worst case. If several platforms might be used, WCET will be estimated for each of them separately. For some real-time systems, WCET calculation

methods have to be *safe* and as *tight* as possible. Safety means that the WCET estimate must be higher than or equal to the actual worst-case execution time. Tightness makes the estimate useful: to avoid over-provisioning processor resources, the estimated WCET has to be as close as possible to the actual WCET.

WCET calculation techniques can be classified into two categories: static and measurement-based methods. Measurement-based methods can miss the worst-case, and static methods overestimate the WCET result and emphasize safety. Static methods analyze the program and possible execution paths to derive WCET results. So in this thesis, we focus on static methods. Static WCET estimation has to be computed at the machine code level, because the timing of processor operations can only be obtained at this level. Moreover, in processors with cache memories, the addresses of memory locations – necessary to analyze the contents of caches – are only known at binary code level.

Information on program control flow is required to calculate tight WCETs. The most basic *flow information* consists in loop bound information (the maximum number of times a loop iterates, regardless of the program input). More elaborate flow information help tighten WCETs, for example by expressing that a given path is infeasible, or that some program points are mutually exclusive during the same run.

Flow information may be obtained by using static analysis techniques or added manually by the application developer through annotations. In both situations, it is convenient to extract or express flow information at the source code level. When using manual annotations, the application developer can focus on the application semantics and behavior, ignoring the compiler and the binary code. When extracted automatically, more flow information can be gathered at source code level than at binary code level because of the higher level of the analyzed language.

Compilers translate high level languages written by programmers into binary code fit for microprocessors. Modern compilers also typically apply hundreds of optimizations to deliver more performance. Some of them are local (i.e. at the granularity of the basic block), they usually do not challenge the consistency of flow information. Other optimizations radically modify the program control flow. As a result, it is usually very difficult to match the structure of the binary code with the original source code, and hence to *port* flow information from high-level to low-level representations. Even when the structure of the binary and source code seem to match, there may be important changes of loop bound information, through optimizations such as loop unrolling or loop re-rolling. Figure 1 (shown in C language for readability, although it will be expressed in compiler Intermediate Representation (IR), or binary code) shows the application of loop unrolling. Loop unrolling in this example replicates the loop body twice in one iteration ($body(i); \rightarrow body(i); body(i+1);$). The structure of the code does not change after the optimization. But the loop bounds are not the same ($100 \rightarrow 50$).

Using optimizing compilers is key to deliver performance. From the point of view of the programmer, compilers are black boxes that take source code as input, and produce binary code. Some compilers can produce dumps of the transformations they applied, but these dumps are very limited. Yet, modern compilers apply hundreds of transformations, some very aggressive, that radically modify the structure of loops

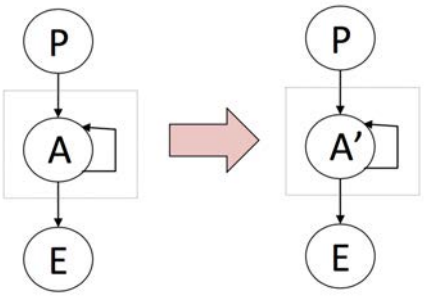
```
for (i=0; i<2*n; i++)
  // MAXITER(100)
{
  body(i);

}
```

(a) Original source code

```
for (i=0; i<2*n; i+=2)
  // MAXITER?
{
  body(i);
  body(i+1);
}
```

(b) Optimized (unrolled)



(c) The structure of original code and unrolled code

Figure 1 – CFG matching and WCET overestimation

(consider unrolling, software pipelining, fusion, tiling, polyhedral transformations...) and functions (inlining, specialization, processing OpenMP directives).

Using the flow information obtained at the source code level, or using best-effort methods for matching source code and binary code may be misleading. In the favorable case, the WCET is "simply" overestimated. Consider the example of Figure 1. The loop on the left has been annotated by the programmer. After optimization, in particular loop unrolling, the code will be similar to the right part of the figure. Both contain a single loop, and a tool could be tempted to match the CFGs and port the flow information to the binary representation. In this particular case, the result remains safe, but precision is lost since the new loop obviously iterates only 50 times at the maximum, whereas the original loop iterates 100 times. On the other hand, loop rerolling (implemented in some compilers – including LLVM [LA04, LLVb] – to reduce code size) results in a increase of the number of loop iterations. Using graph matching for such loops would result in underestimated WCETs, that jeopardize the system safety.

So now we are facing the following issues:

- In order to derive better performance, modern compilers usually include optimizations. The quantity and type of the optimizations vary depending on the compilers.
- If the compiler optimizations are applied, the flow information may be modified.
- When the flow information is modified, if we do nothing, better case is that the WCET is overestimated but safe; worse case is that the result is unsafe (*e.g.* loop

rerolling increases the number of loop iterations), or even cannot be got (*e.g.* loop unswitch and loop tiling add new loops whose loop bounds are unknown for WCET estimation).

- We have mentioned that graph matching may be unsafe as well.
- So a big problem comes: we cannot estimate a safe and as precise as possible WCET result with modern optimizing compilers.

The solution we proposed is to trace flow information in the optimizing compilers. Annotations are essential flow information to make WCET analysis more precise. So the traceability of annotations is the objective of our thesis.

The thesis is funded by the project W-SEPT[1] which is a collaborative research project focusing on the precise estimation of the worst-case execution time and the identification and traceability of semantics information through the compilation flow from high-level language to C level and finally to binary level. The project is funded by ANR[2] under grant ANR-12-INSE-0001, and supported by the competitiveness clusters Aerospace Valley[3] and Minalogic[4]. Our work is also partially funded by COST Action IC1202: Timing Analysis On Code-Level (TACLe)[5].

Contribution of the thesis

In this thesis, we propose a framework to systematically transform flow information from source code level down to binary code level. The framework defines a set of formulas to transform flow information for standard compiler optimizations. What is crucial is that transforming the flow information is done within the compiler, in parallel with transforming the code. There is no *guessing* what flow information have become, it is transformed along with the code they describe. In case the transformation is too complex to update the information, we may have the option to drop the information if the information is not necessary. In this way, we can guarantee that the result is safe, even though it will probably result in a loss of precision.

Our transformation framework is designed to transform flow information as expressed by the most prevalent WCET calculation technique: Implicit Path Enumeration Technique (IPET) [LM95]. More precisely, flow constraints are expressed as linear relations between execution counts of basic blocks in the program control flow graph. As shown later in the thesis, the framework is general enough to cover all typical optimizations implemented in modern compilers.

The proposed framework was integrated into the LLVM compiler infrastructure. For the scope of this thesis, although formulas are more general, our experiments in LLVM will concentrate on *loop bounds* as sources of flow information.

[1] http://wsept.inria.fr
[2] http://www.agence-nationale-recherche.fr
[3] http://www.aerospace-valley.com
[4] http://www.minalogic.org
[5] http://www.cost.eu/COST_Actions/ict/Actions/IC1202

Usually, programs spend most of the execution time in loops and in (recursive) functions. So determining loop bounds and function depths is an essential task for WCET estimation. Loop optimizations and function inlining in modern compilers make this task a challenge. Fortunately, in our implementation, loop bounds are traced carefully and all loop optimizations in LLVM are supported. Besides, inlining optimization is also included. Therefore, our framework and implementation accomplish this essential and important task, and obtain the safe and tight WCET even with the compiler optimizations.

Our experimental results show that LLVM optimizations significantly reduce estimated WCETs.

Structure of the thesis

The contents of this thesis is organized as follows:

Chapter 1 gives an introduction to WCET, WCET estimation methods, WCET estimation tools and prototypes. Then, the most common WCET static analysis technique, IPET is described. At the end of this chapter, annotation languages are presented.

Chapter 2 describes the theoretical foundations required by our work. Then the main contribution of this thesis, a transformation framework, is proposed. Our transformation framework can trace flow information with compiler optimizations independently of the compiler framework. A summary of our supported compiler optimizations and their corresponding rule sets is presented in this chapter. At the end, we introduce and compare the research works related to our transformation framework.

An implementation of the proposed transformation framework within the LLVM compiler infrastructure is presented in Chapter 3. We present the overview of LLVM and then our traceability method and the corresponding modification of LLVM.

We provide experimental setup, results and their analysis in Chapter 4. Through the results and analysis, we can derive that:

- Flow information can be traced by our transformation framework during the compiler optimizations.
- With our framework and implementation, we can estimate safe WCET.
- Estimated WCET can benefit from compiler optimizations.

Finally, we conclude with a summary of the thesis contributions and plans for future work.

Chapter 1

WCET Estimation Techniques

This chapter gives an overview of Worst-Case Execution Time (WCET) estimation techniques and flow information transformation. After a generic introduction about WCET calculation techniques and tools in Section 1.1, the static WCET estimation method – Implicit Path Enumeration Technique (IPET) is described in Section 1.2. Afterwards, the different existing annotations forms and the corresponding extraction methods are presented in Section 1.3.

1.1 Worst-Case Execution Time Analysis

"Deadlines" are the specified response time constraints which should be guaranteed by real-time systems. By the consequence of missing a deadline, real-time systems can be classified into the following three categories:

Hard Ensure that all deadlines are met. Missing a deadline is a total system failure.

Firm Infrequent deadline misses are tolerable, but may affect the quality of service. The computation after its deadline is obsolete.

Soft Deadline misses are tolerable, but not desired.

Hard real-time systems have to fulfill strict timing guarantees, otherwise catastrophic consequences may be caused. So to avoid this happening, the worst-case execution time (WCET) of the program needs to be known.

1.1.1 Worst-Case Execution Time and WCET estimation

WCET is an essential element for hard real-time systems. However computing the execution time of a program in the worst case is challenging. If the worst-case input for the program were known, we could derive a reliable guarantee based on the worst-case execution time. Unfortunately, in general this worst-case input is unknown and hard to derive.

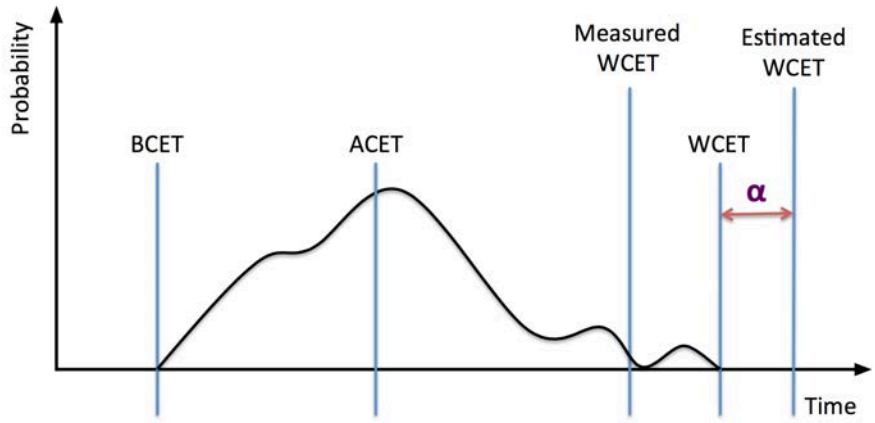


Figure 1.1 – Distribution of execution time and basic notions of timing analysis systems

The relevant typical terms used to describe the execution time of a program are depicted in Figure 1.1. The curve represents the probability of different execution times of a program with different input data and initial processor states. Usually, the curve applies to a given hardware/OS pair, and the execution times usually vary widely depending on the different program inputs. The best-case execution time (BCET) is the shortest execution time of the program. And worst-case execution time (WCET) is the longest execution time. Average-case execution time (ACET) lies between the BCET and WCET. ACET depends on the distribution of the execution time of a program. The average of the execution time observed for one input data set is called ACET. Average performance and worst-case performance are the most used.

Since worst-case performance is so important to hard real-time systems, how can we get WCET? WCET calculation techniques can be classified into two categories: static and measurement-based methods [WEE+08].

Static methods abstract the program first, and then analyze the abstraction of the program and find all possible paths. Then the methods analyze the set of possible execution paths, and do not rely on execution of the program on real hardware or simulator which often needs complex equipment for the target system. They derive upper bounds of the WCET from the program structure and a model of the hardware architecture, and maybe together with some annotations. Annotations are the external information that are given explicitly by static analysis tools or program developers or users. More details about this category of methods are presented in [WEE+08].

The estimated WCET in Figure 1.1 is derived by static methods. The estimated WCET is overestimated, because during the abstraction, the behavior of the processor cannot be predicted accurately, and a pessimistic result is usually used.

On the other hand, measurement-based methods use end-to-end measurements on the given processor or a cycle-accurate simulator, with a set of possible inputs data, in search for the input that exercises the longest execution path. The advantages for these methods are that they do not need to model hardware architecture and thus can be easy

to be applied to other new target systems. Besides, for complex programs and complex target systems, they are simpler to be applied. The measured WCET in Figure 1.1 is derived by these methods. In general, the WCET is underestimated, unless the worst case input is exercised by the measurements.

In Figure 1.1, α is the difference between actual WCET and estimated WCET ($\alpha = estimated\_WCET - actual\_WCET$). It can be used to explain the two main criteria of WCET estimation: safety and precision. Safety means that the estimated WCET must be higher than or equal to the actual WCET, *i.e.* α should always be positive. This is essential for hard real-time programs to avoid severe damages. Precision means that the estimated WCET has to be as close as possible to the actual WCET. *i.e.* α should be as small as possible. Actually, α can be considered as explicit precision. A lack of precision may lead to a waste of hardware resources.

Static methods emphasize safety. These methods are based on the target hardware model that can capture the behavior of the processor. By design, static methods are guaranteed to identify the longest feasible execution path. And by using this longest feasible path it can produce the bound which can guarantee that the actual execution time will not exceed the bound, and therefore, are safe. However, the necessity for the specific model of hardware architecture and the possibly overestimated WCET estimate are the price paid for the safety. Nowadays, modeling hardware architecture and processor behavior is still a main technical problem.

The disadvantage of measurement-based methods is distinct: these methods may miss the actual worst case and may underestimate WCET, unless the target systems or test programs are simple enough, or all possible execution paths can be measured. So the underestimated WCET is impossible to be used in hard real-time systems. So we do not consider measurement-based techniques in this thesis.

Besides, there are some other timing analysis techniques, *e.g.* hybrid measurement-based analysis and probabilistic timing analysis.

Hybrid measurement-based analysis is similar to static methods. The differences are that hybrid timing analysis does not need to model hardware architecture, it derives execution times of small program segments by using measurement [KPW04, GBEL10, BMB10]. Then it combines them with static methods. The WCET estimated by this method is generally more accurate compared with static methods. However, there are kinds of drawbacks. For example, there is a possibility of underestimation; it adds an overhead which may disturb the accuracy and slow the program. The WCET analysis tool RapiTime [Rapb, Rapa] is an example.

Probabilistic WCET analysis [BCP02, BCP03] is a method combining static and measurement analysis in a probabilistic framework. The probability distribution of the WCET of a code fragment can be determined by this method. It extracts the program into a syntax tree in which the leafs are basic blocks and the inner nodes are the sequential, conditional and iterative parts. Then it uses a timing schema to represent the different types of nodes in the syntax tree. A measurement approach can be used to record the actual execution time of each nodes. With these information, a WCET result can be derived.

1.1.2 WCET analysis tools and prototypes

In this subsection, we summarize the main WCET analysis tools and prototypes for the moment.

aiT aiT WCET Analyzer is the WCET analysis tool of AbsInt which is a software-development tools vendor based in Saarbrücken, Germany. The purpose of aiT [Abs, FH04, FHF07] is to obtain tight bounds for the WCET of tasks in real-time systems. It directly analyzes binary executables and statically analyzes a task's intrinsic cache and pipeline behavior to compute correct and tight upper bounds for the WCET.

Bound-T The Bound-T timing analysis tool [Tid, HS02] provided by Tidorum Ltd computes an upper bound for the execution times fo programs by using static analysis of the machine code. Optionally, the tool can also get bounds on the stack usage of the subroutine, including called functions.

Chronos Chronos [oS, LLMR07] is an open-source static WCET analysis tool developed at National University of Singapore (NUS). Chronos models various architectural features and their interactions for WCET analysis. By analyzing binary code, it constructs control flow graph and extracts flow constraints. With these flow constraints, processor model, user configuration, additional flow information, Chronos determines an upper bound on the execution time of a program.

Heptane Hades Embedded Processor Timing ANalyzEr (Heptane) [CP00, IRI] is an open-source static WCET analysis tool designed by IRISA, Rennes (part of W-SEPT). Through statically analyzing source and/or binary code[1], Heptane extracts control flow graph, loops, basic blocks and so on. With these information and loop annotations, it analyzes cache and computes the upper bounds on the execution time for many cache architectures.

OTAWA Open Tool for Adaptive WCET Analyses (OTAWA) [BCRS10, TRA] is a static WCET analysis tool developed by the TRACES team at IRIT labs, University of Toulouse, France (part of W-SEPT). OTAWA proposes abstract layers to make the analyses independently from the hardware and from the instruction set. By analyzing binary programs, the extracted flow information combining the information of the target hardware and annotations are used for the final WCET static analysis.

RapiTime RapiTime [Rapb, Rapa] is an automated measurement-based WCET analysis tool developed by Rapita Systems Ltd. RapiTime measures the WCET result by running the real-time programs with a suite of tests. The users need to provide test

[1]Source code needs to be compiled into binary code and Heptane analyzes the binary code finally.

suite to and can provide annotations in the code to guide the measurement. Benefiting from the measurement-based method, RapiTime does not rely on a model of the processor, and it can handle complex advanced architecture.

SymTA/P The purpose of SYMbolic Timing Analysis for Processes (SymTA/P) [oCNE] from IDA, TU Braunschweig is to obtain upper and lower execution time bounds of C programs by modeling and analyzing process behavior using execution cost intervals. The program structure and the execution context is considered in this tool.

SWEET SWEdish Execution Time tool (SWEET) [RtiV, Lis14] is a research prototype provided by Mälardalen Real-Time Research Center (MRTC). It can translate different code formats into their intermediate format ALF [GEL+09]. With ALF, SWEET analyzes and derives flow information automatically. At the end, it calculates safe bounds on the possible executions of a program.

T-CREST Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST) [tcr, PPH+13] is a time-predictable system built collaboratively by industrial organisations and research and development organisations. It aims at the derivation of the WCET of the hard real-time space applications with multi-core technology. For this purpose, T-CREST proposes solutions on both the hardware (*e.g.* time-predictable caching) and the compiler infrastructure (*e.g.* LLVM compiler infrastructure and WCET aware optimizations). At the end, T-CREST provides a WCET analyzable multi-core system with high performance.

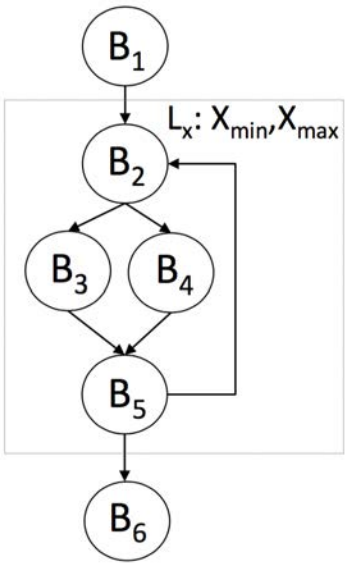
1.2 Static WCET Calculation Using IPET

The WCET calculation method used in the thesis is the most common technique, named IPET for *Implicit Path Enumeration Technique* [LM95, PS97]. This method operates on control flow graphs (CFG), extracted from binary code. IPET models the WCET calculation problem as an *Integer Linear Programming (ILP)* [GN72] formulation.

1.2.1 Integer Linear Programming (ILP)

At first, Linear Programming (LP) should be introduced. LP, also called linear optimization, is a method to process various linear inequalities relating to the requirements and find the best outcome of this mathematical model under these conditions. In a linear program, there are variables, constraints, and an objective function. The variables stand for numerical values. Constraints are linear expressions and used to limit the values to a feasible region. The objective function should also be linear in the variables. Its declaration consists of one of the keywords *minimize* or *maximize*. It defines the quantity to be maximized or minimized subject to the constraints.

For ILP, it adds the requirements that some or all of the variables are restricted to be integers. ILP can be expressed in the following canonical form:



Objective function

$$Maximize \sum_{i \in CFG} f_i \times T_i$$

Structural constraints

$$\begin{aligned} f_1 &= 1 \\ f_{1\_2} + f_{5\_2} &= f_{2\_3} + f_{2\_4} = f_2 \\ f_{2\_3} &= f_{3\_5} = f_3 \\ f_{2\_4} &= f_{4\_5} = f_4 \\ f_{3\_5} + f_{4\_5} &= f_{5\_2} + f_{5\_6} = f_5 \\ f_{5\_6} &= f_6 \end{aligned}$$

Additional constraints

$$\begin{aligned} f_2 &\leq X_{max} \\ f_2 &\geq X_{min} \\ f_3 &\leq 2 \times f_4 \end{aligned}$$

Figure 1.2 – CFG and WCET calculation using IPET

Variable $x_i \in int \quad (i = 1, 2, \ldots, n)$

Constraint $\sum_{i=1}^{n} a_{ij} \times x_i \leq b_j \quad (j = 1, 2, \ldots, m)$

$x_i \geq 0 \quad (i = 1, 2, \ldots, n)$

Objective function Maximize/Minimize $\sum_{i=1}^{n} c_i \times x_i$

a_{ij} and b_j are the parameters of the constraints, and c_i are the parameters of the objective function. All of them are integer constants.

1.2.2 Timing analysis with IPET

An example CFG is depicted in the left part of Figure 1.2. The example program includes one loop, depicted by a rectangular box. Notations X_{min} and X_{max} state that the loop iterates at least X_{min} times, and at most X_{max} times. They are used in additional constraints. (More information in Section 2.1.2)

The right part of Figure 1.2 depicts the ILP system used to calculate the WCET. Every basic block i has a worst-case execution time, denoted as T_i, and considered constant in the ILP system. Calculating the WCET is done by maximizing the objective function, in which f_i is variable and represents the execution count of basic block i. The control flow is subject to structural flow constraints, that come directly from the

structure of the CFG and are generated automatically. From top to bottom, the first one states that the entry point to be analyzed is executed exactly once. The next constraints state that the execution count of a basic block is equal to the sum of the execution counts of its incoming edges, as well as outgoing edges, where f_{i_j} represents the execution count of the edge from node i to j.

Finally, additional constraints specify flow information that cannot be obtained directly from the control flow graph. The first kind of additional information is loop information ($f_2 \leq X_{max}$ and $f_2 \geq X_{min}$ in the example). It gives the maximum number of iterations for loops, and is mandatory for WCET estimation. Some other linear constraints such as $f_3 \leq 2 \times f_4$ may also be specified to constrain the relative numbers of executions of basic blocks in the CFG. Additional constraints come from the semantics of the programs and cannot be derived easily, so they may be inserted manually by the programmer, through annotations, or be obtained automatically using static analysis tools.

1.3 Flow Information and Annotation

Previous section mentioned that static analysis methods may need some annotations (*e.g.*, loop bounds, infeasible paths and so on) to derive upper bounds for the execution times of programs on a given platform. In fact, WCET estimation usually needs manual annotations or assertions to define essential information. Information on the flow of control of applications improves the tightness of WCET estimates. Beyond loop bounds, which are mandatory for WCET calculation, examples of flow information include infeasible paths, contextual information, or other properties constraining the relative execution counts of program points. An annotation language is used to annotate the flow information and make the information available to the subsequent WCET analysis. The following seven pivotal points decide the design of an annotation language and have an impact on its usability. They are related to both the WCET analysis tools and the target programs.

- The need for annotations
- The form of annotations
- The content of annotations
- The supported language
- The placement of annotations
- The level of annotation
- The method of annotations addition

1.3.1 The need for annotations

The WCET analysis methods and tools vary widely. But many of them need annotations for precise WCET estimation. Sometimes, the annotations improve even the efficiency of the estimation.

1.3.2 The form of annotations

Normally, the form of annotations is special to its own WCET analysis tool. It relates to the placement, the level, even the method of addition. For example, for external separate annotations, Extensible Markup Language (XML) [BPSM+98], as a simple and universal format, is used widely to represent annotations.

One example is an XML-based representation [PM14] proposed by Parsa et al. They use this XML-based annotation to store the information from the analysis of program and to provide to other WCET analysis tools for precise WCET estimation.

FFX [BCdM+12] is another XML-based annotation format. Benefiting from the XML standard, it is easy to create, understand and use among different WCET tools.

1.3.3 The content of annotations

Theoretically and ideally, the annotations should contain all flow information (loop information, infeasible paths, contextual information and so on). However, considering the limitation of annotation format, the need of WCET analysis tools, and difficulty of flow information acquisition, annotations should choose the appropriate content. But for almost every annotation, loop bound is the essential part.

1.3.4 The supported language

Most annotations support a single programming language (based on WCET analysis tools, and for now C language is supported most widely). There are a few annotations that can support multiple languages.

1.3.5 The placement of annotations

There are two primary ways to keep annotations: inside the source code or in a separate file. None of them is consistently superior to the other. Normally, the annotations are usually added inside the source code when the annotations must be added manually. Because in this way, it is much easier and less error-prone. On the contrary, when the annotations are derived by the static analysis tools, writing into a separate file is more convenient for both the analysis tools and WCET estimation tools.

1.3.6 The level of annotations

Annotations can be added at high-level source code or low-level machine code.

The advantage of addition at machine code level is the convenience of usage. Because the WCET estimation is also at machine code level, the annotations can be used directly.

The addition of annotations at this level is usually through two methods. One is to read and analyze machine code. This is difficult for the programmers, users and the analysis tools. Another is to map with source code. This is typically non-trivial. When the optimizations are applied, this becomes difficult and error prone. One method to maintain the mapping between source code and machine code is to define a set of language constructs: *anchors* [KKP+07]. which can be recognized after compilation.

Compared with machine code level, adding annotations at source code level is easier for both programmers and analysis tools. And these annotations are easier to understand and verify. So, normally, source code level annotations are chosen in most cases.

1.3.7 The method of annotations addition

Annotations can be obtained via two basic methods: static analysis or annotations added by the application developers or users.

Manual annotations are an easy and convenient way to assist non-perfect analyses. Usually, they are added by the users who know the code well, *e.g.* the program developers. However, manual annotations are potentially error-prone and may yield incorrect WCET estimates.

In order to get the annotations automatically, the static analysis tools are required. With the development of the techniques of flow information extraction, more and more annotation languages use the automatic additional methods.

At the same time, more and more methods of flow information extraction are proposed. Here are several examples:

Gustafsson et al. [GESL06] propose a method called abstract execution. This method can automatically calculate loop bounds and infeasible paths. Their method can calculate nested loop bounds. They verify their method by using Mälardalen benchmark suite.

Blackham et al. [BLH14] propose an infeasible paths detection method called Trickle. This method analyzes the binary programs and detects infeasible paths within the CFG to refine WCET estimations.

Holsti et al. [HGKL14] use the program-representation language ALF to combine two analysis tools: Bound-T and SWEET. The combination can resolve the analysis of dynamic branches at binary code level. They can generate an annotation file containing the analyzed control flow information. This annotation file is used to guide the further WCET estimation.

Bonenfant et al. [BdMS08, dMBBC10] develop a static loop bound analysis tool called oRange. The tool is based on flow analysis and abstract interpretation, and can extract and provide loop bound values or equations, non-recursive function calls and other flow information. The provided flow information can be used in static WCET analysis.

There are still many automatic methods concentrating different kinds of flow information: branch constraints detection and exploitation [HW02], loop bounds extraction [LCFM09] and so on.

Nowadays, the hybrid methods are used by some WCET tools. They base on automatic extraction, and use manual addition to fix the unobtainable part, *e.g.* aiS and Bound-T annotation language (introduced in the next subsection).

1.3.8 Summary of annotation languages

Here, we give a summary of WCET annotation languages. More detailed WCET annotation languages description and systematic comparison are presented in [KKP+07, KKP+08, KKP+11].

Real-Time Euclid [KS86] is one of the first language designed specifically to feature annotations for timing analysis in real-time systems. Only the specification of loop bounds in *for* loops are supported.

Another early annotation is proposed by Park et al. in [PS90, Par93, Cha94]. They define the information description language (IDL) as an interface language for users to provide and express flow information. This approach uses IDL to perform the mapping from the object code to the source code. Path patterns of explicit execution order can be expressed by this annotation, and this is also its advantage.

WCETC [Kir02] is designed as a new programming language. It is based on ANSI C and extends it with new grammar. Benefiting from the new grammar, additional flow information including loop bounds or infeasible/feasible paths can be specified directly in the source code. With the additional flow information is used to estimate the WCET. The advantage of this annotation is that it can annotate flow information exactly at the location where the program should be described.

The annotation language of Heptane [IRI] is designed to provide loop bounds. Its difference is that these annotations can be provided in two ways: add loop bounds inside each loop in the C source code; provide an external XML-based annotation file in which the loop bounds are given and the order of loops should be the same as in the compiled binary code.

Mok et al. [Che87, MACT89] propose the Timing Analysis Language (TAL). TAL is an integral part of the timing analysis system. It consists of multiple tools. The *annotation tool* can analyze C source code and automatically generate the annotations of the C code with default assumptions about the program's behavior. Then a modified C compiler translates annotated C programs to annotated assembly programs, because *timetool* which calculates the execution time works only on assembly code. The advantage of this annotation is that it may contain arbitrary calculations.

The Bound-T applies a data-flow analysis to automatically compute loop bounds. When the loop bounds could not automatically be bounded, it involves user-assistance. These automatic and manual information are stored in a separate file. Then all these information are used for the WCET estimation.

aiS is the annotation language of aiT. aiT applies an analysis to automatically calculate flow information. It also needs additional annotations provided by users in the aiS format called AIS file. The AIS file needs to provide not only loop bounds, but also recursion bounds, even the targets of calls and branches.

Both of these two annotation language are the language of commercial WCET analysis tools, and they use hybrid WCET annotation addition method.

1.4 Summary

This chapter describes the notion of WCET and WCET analysis methods, then lists the WCET analysis tools and prototypes. Then we introduces the most common WCET static analysis technique: IPET. At the end of this chapter, another important notion in this thesis is presented: annotations. Through this chapter, we know that WCET is essential for hard real-time systems. However, the optimizations in modern compilers make the WCET estimation complex. Because, the optimizations break the link between the annotations and the code, the annotations cannot be used directly. Our thesis focuses on this issue, and fortunately, we propose a solution: transformation framework of flow information. In the next chapter, we will present our transformation framework.

Chapter 2

Transformation Framework

In this chapter, the theoretical foundations required by our transformation framework are introduced firstly in Section 2.1. We need flow information to calculate the WCET bound. The flow information is translated into linear constraints (shown in Section 2.2). However, some optimizations may have an effect on these constraints, furthermore affect the calculation of WCET. So in order to get more precise estimated WCET bound after the optimizations, we need a method which can declare the flow information transformation and update the constraints.

So in Section 2.3, the main contribution of the thesis is introduced. we propose a transformation framework that conveys flow information from source code level to machine code level with the compiler optimizations. The transformations are expressed in an abstract way, independently of the compiler infrastructure in which they will be integrated. In this part, we introduce the transformation framework with the following aspects: transformation rules, the manipulation and its influence on WCET estimation. Afterwards, an overview of the transformation framework is provided in Section 2.4.

Then in Section 2.5, the code optimizations performed by the compiler and the advantages of these optimizations are described according to their impact on the control flow and WCET analysis. For each compiler optimization, our transformation framework, defines a set of formulas, that rewrite available flow constraints into new constraints according to the code transformation. Then these new constraints can be used for the WCET estimation.

At the end, in Section 2.6, the related work to the traceability of flow information and WCET estimation with compiler optimizations are described.

2.1 Flow Information

Flow information is the information of the control flow of a program. Flow information can be expressed explicitly by the control flow graph (CFG), also by additional information provided externally.

Through analyzing the CFG, we can get structurally feasible paths. When the compiler optimizations are applied, the CFG is usually modified. These flow information

```
for(int i=0; i<n; i++)
{
  if(a[i]>0)
    b[i]=b[i]+i;
  else
    b[i]=a[i]+n;
}
```

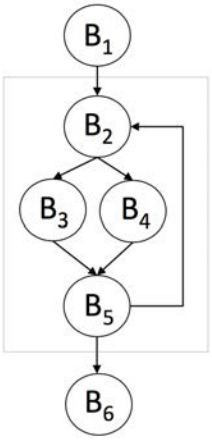


Figure 2.1 – Example of control flow graph (CFG)

can be updated automatically according to the new CFG. Unfortunately, the relation between the external information and the CFG of the source code is broken by the optimizations.

The rest of this section explains the notions and definitions about flow information.

2.1.1 The program representation

Our transformation of flow information operates on the program control flow graph (CFG) [All70]. For presentation clarity, we will concentrate in this thesis on a single CFG, although our transformation framework supports multiple functions and function calls. The CFG is important, because it is used in most compilers, and is essential to many compiler optimizations and static analysis tools.

At first, the definition of basic block is given because it is used in the definition of CFG.

Definition 1 A **basic block** is a straight-line piece of the code within a program with only one entry point and only one exit point, *i.e.* without any jumps except the last instruction or jump targets except the first instruction.

Here, the code can be assembly code, intermediate representation or some other sequence of instructions.

Definition 2 A **CFG** is a representation using graph notation. A CFG is a (possibly cyclic) directed graph made of a set of nodes $\mathcal{N}$ representing basic blocks, and a set of edges $\mathcal{E}$ representing possible control flows between basic blocks.

In the example program of Figure 2.1[1], we have:

[1]The CFG is correct only if $n > 0$.

$$\text{CFG} = \{\mathcal{N}, \mathcal{E}\}$$
$$N = \{B_1, B_2, B_3, B_3, B_4, B_5, B_6\}$$
$$\mathcal{E} = \{B_1 \to B_2, B_2 \to B_3, B_2 \to B_4, B_3 \to B_5,$$
$$B_4 \to B_5, B_5 \to B_2, B_5 \to B_6\}$$

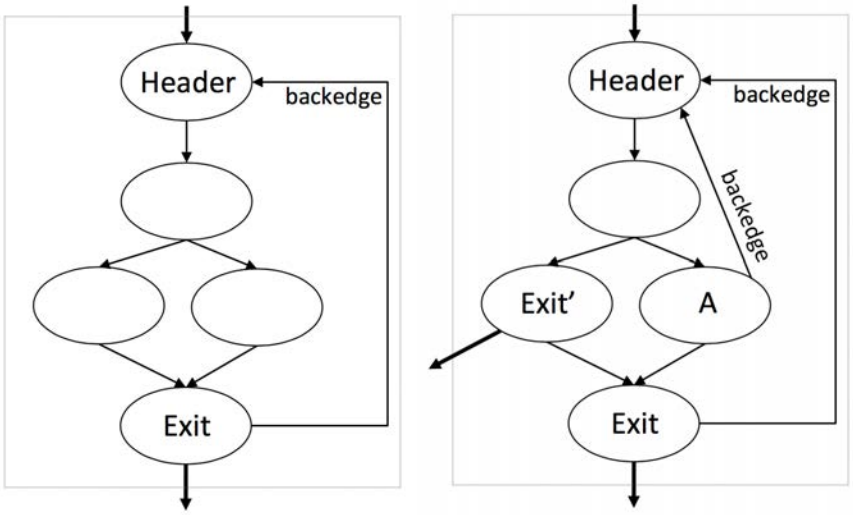
2.1.2 Loop description

During the flow information, loop bound is the most important notion. Because loop bound information is the necessary information to derive WCET estimate.

Firstly, we need to define *strongly connected component* which is the most general looping structure.

Definition 3 A **strongly connected component** of a flow graph $G = (N, E)$ is a subgraph $G_s = (N_s, E_s)$, in which there is a path that includes only edges in E_s from every node in N_s to every other node in N_s.

Definition 4 A **loop** is a strongly connected component of the CFG.



(a) Single loop (b) Multiple loop

Figure 2.2 – Loops

Then, we introduce the following properties of loops:

Entry nodes/Loop headers are the nodes into the loop from outside. They dominate all nodes in the loop[2]. In this thesis, we only consider the loops with unique loop header (as shown in Figure 2.2).

[2]Node A dominates node B if every path from the entry node to B must go through A.

Exit nodes are nodes with edges going to the nodes outside of the loop. There can be several exit nodes, *e.g.* in the example of Figure 2.2b, there are two exit nodes: the exit node similar to the one of the single exit node loop and the exit node $Exit'$ which can be brought by instruction *break*.

A backedge is a part of the loop. It is an edge from node A to node B if B dominates A ($A, B \in$loop). A loop can have more than one backedge. For example, in the example of Figure 2.2b, besides the backedge similar to the one of the single backedge loop in Figure 2.2a, edge $A \rightarrow Header$ is another backedge which can be created by instruction *continue*. Optimization *Loop Simplify*[3] can turn the loop into single backedge loop. For single backedge loop, backedges can be used to represent the iteration count of the loop.

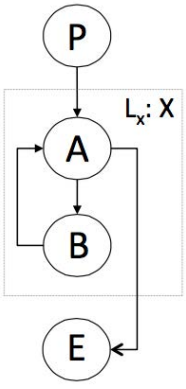
Figure 2.2 shows two typical loop examples. The left one is a typical loop with single entry node and single exit node. The right one is a loop with multiple exit nodes and multiple backedges. In fact, a loop can have multiple entry nodes, exit nodes or multiple backedges, while in this thesis, we consider only reducible loops: single entry node, one or multiple backedge and one or multiple exit nodes.

```
i=0;
while(i<X){
  a[i]=a[i]+b;
  i++;
}
```

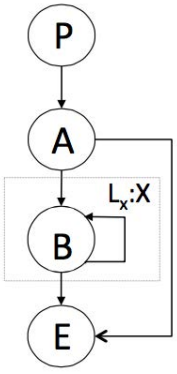
(a) Source code of top tested

```
i=0;
if(X>0){
  do{
    a[i]=a[i]+b;
    i++;
  }while{i<X}
}
```

(b) Source code of bottom tested



(c) CFG of top tested loop



(d) CFG of bottom tested loop

Figure 2.3 – Different location of test nodes.

[3]Loop simplify optimization transforms natural loops into simpler ones. For example, it guarantees the loop with a single edge from outside of the loop to the loop header; with a single backedge and so on. In this way, further optimizations are simpler and more effective to apply.

Figure 2.3c and Figure 2.3d show two different CFG of loops. Actually, these two loops do the same function (refer to the source code Figure 2.3a and Figure 2.3b). Node B in both loops is executed X times. But for node A in Figure 2.3c, it is executed $X + 1$ times. The loop bounds of both loops are X. So for the bottom tested loop, the execution count of each node in the loop body equals to loop bounds. The test node in top tested loop is executed one more time than loop bound. So for uniformity, we define loop bounds as:

Loop bounds are the maximum number of executions of the nodes in the loop body except the node(s) testing the loop exit[4].

A Local loop bound represents the maximum number of iterations of a loop for each entry.

A Global loop bound is an upper bound on the execution count of a loop in a whole program. It is considered in nested loops.

`for(i=0; i<10; i++)`
`  a[i]=a[i]+1;`
(a) Loop A
	`for(i=0; i<n; i++)`
	`  a[i]=a[i]+1;`
	(b) Loop B

Figure 2.4 – Example of Loop Bounds

Sometimes, the loop bounds are not fixed, they can be intervals. For example, in Figure 2.4, the loop A has a fixed loop bound 10. For loop B, its loop bound depends on the value of n. When $10 \leq n \leq 20$, the loop bounds of loop B range from 10 to 20. We call the endpoints of the loop bound interval (a, b for interval [a,b]) the lower loop bound (for a) and the upper loop bound (for b). So for loop B, its lower loop bound is 10 and upper loop bound is 20. In this thesis, the lower loop bound is expressed as LB_{min} and the upper loop bound LB_{max}.

When we mention global loop bound, we refer to nested loop. Now we give the definition of this notion.

Definition 5 **Nested Loops** are a set of loops in which each loop except the outermost one is within the body of another.

Definition 6 **Perfect loop nest**[5] means that except the innermost loop, each loop contains only another loop in the nest.

The proposed framework for traceability of flow information in the next chapter assumes reducible nested loops. Information on loop nesting is captured through the *Loop Scope* data structure.

[4]If the test node is the only node in the loop, the loop bound is its maximum execution number.

[5]Perfect loop nest is used in some loop optimizations, *e.g.* loop interchange.

Definition 7 **Loop Scopes** are the code boundaries of loops.

The information is made of a set of pairs $\langle L_o, L_i \rangle$ with L_o and L_i as loops. L_i is the inner loop and is completely nested in the outer loop L_o. This data structure is useful because some compiler optimizations involve multiple loops (e.g. loop interchange), their maximum number of iterations have to be modified jointly.

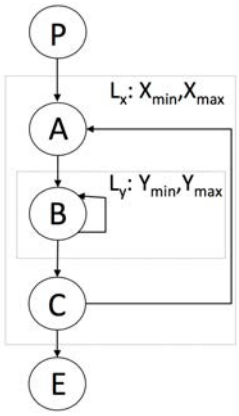


Figure 2.5 – Running example with nested loops

The *Loop Scope* data structure for our running example depicted in Figure 2.5 contains:

$$\text{LoopScope} = \{\langle \_, L_x \rangle, \langle L_x, L_y \rangle\}$$

with "_" denoting the absence of enclosing loop for the outermost loop.

```
for(i=0; i<10; i++)
  for(j=0; j<10; j++)
    a[j,i]=a[j,i]+1;
```
(a) Loop Nest A

```
for(i=0; i<10; i++)
  for(j=0; j<i; j++)
    a[j,i]=a[j,i]+1;
```
(b) Loop Nest B

Figure 2.6 – Local/Global Loop Bound

When the loop is nested, the local bounds of all the loop bounds involved in the nest are estimated separately as if there was no nesting, but the global loop bounds of the innermost loop need a summation of the maximum iteration number in each iteration of the outer loop. Using local loop bound and global loop bound has no difference except several special cases *e.g.* triangular loops. For example, considering the loop nest A in Figure 2.6, the local loop bounds of the outermost and innermost loop are both 10, and the global loop bound of the innermost loop body $a[j,i] = a[j,i] + 1$ is 100 which is equal to the product of local outermost loop bound and local innermost loop bound. However, for the loop nest B, the situation is different. The local loop bound is the same as nested loop A. The global execution count of the innermost loop body is 45, not 100.

```
if(x>5){
   y=2;
   a=3;
   z=a+y;
}
else
   y=-2;
if(x>0)
   z++;
else{
   z--;
   b=4;
}
```

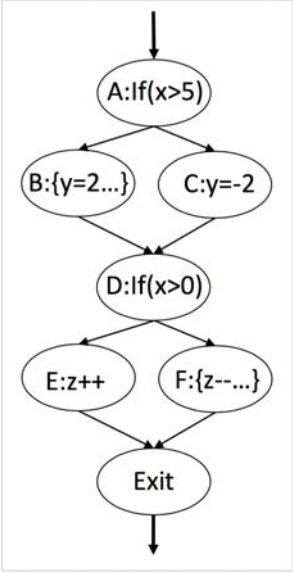


Figure 2.7 – Example of infeasible paths

Knowledge of global loop bounds can tighten WCET estimates. However, the obtention of global loop bound is a challenge, and most WCET analysis tools do not support it. Our transformation framework and our implementation support local loop bound. And in the rest of the thesis, when we mention loop bounds, it should be considered as local loop bound by default.

2.1.3 Infeasible paths

Definition 8 **Infeasible Paths** [Kou96, APT00, GEL06] are the paths which are executable according to the CFG, but not feasible when considering the semantics of the program, the context and possible inputs.

Dependencies are the primary reason for infeasible paths. The classical example of an infeasible path is two consecutive if-then-else structures with interdependent conditions. The example in Figure 2.7 illustrates a very simple example with an infeasible path. The true branch B of the first conditional statement and the false branch F of the second conditional statement are in conflict. Because when the true branch B is taken, *i.e.* x is larger than 5 and also larger than 0, the false branch F is never taken in this situation. So the path $A-B-D-F$ is never taken and it is an infeasible path.

Another case is that when the variable x is an input, and if its input value is 2, the path $A-B$ is infeasible. This kind of infeasible path belongs to input-sensitive infeasible paths.

Information on infeasible paths is not a mandatory information for WCET estimation, just can make WCET estimates tighter. For example, in Figure 2.7, without the information of infeasible paths, the WCET analysis tools should use $A-B-D-F$ to

estimate WCET, because basic blocks B and F have the longer execution time. With infeasible paths, we take $A-B-D-E$ instead of $A-B-D-F$, and derive tighter WCET result.

2.1.4 Contextual information

Normally, a function is designed to be called in different contexts. So the flow information in the function may be different for different calls. For example, a loop in a function can have different bounds for different calls; conditional statements in a function can have different infeasible paths with different input for different calls. This kind of information is called contextual information [BCRS10, CMPVR14].

Similarly to infeasible paths, contextual information is of help to tighten WCET estimation, not indispensable.

2.2 Contents of the Transformation Framework

2.2.1 Representation of flow information

In the first place, we introduce the flow information which the transformation framework operates on. Flow information is available in the following forms:

- Loop bounds, for every program loop:

$$Loopbounds = \{\langle L_x, \langle LB_{min}, LB_{max}\rangle\rangle\}$$

  with L_x the loop identifier and LB_{min} and LB_{max} denoting respectively the minimum and maximum number of iterations for loop L_x, and this for each entry in L_x.

- Infeasible paths, for each infeasible path in the program:

$$Infeasiblepaths = \{i - j - \cdots - k\}$$

  with i, j, k the basic blocks consisting of the infeasible paths.

  For the representation of infeasible paths, we should express the infeasible path as short as possible. For example, as shown in Figure 2.8, there are more than two consecutive conditional tests. When only $A-B-D-F$ is infeasible path, path $A-B-D-F-G-H$ and $A-B-D-F-G-I$ are infeasible. We should express them as $A-B-D-F$, because $A-B-D-F$ is the shortest infeasible path.

- Additional flow constraints that are linear relations on execution counts of basic blocks (f_i):

$$Constraints = \{C_l + \sum_{i \in CFG} C_i \times f_i \;\; op \;\; C_r + \sum_{j \in CFG} C_j \times f_j\}$$

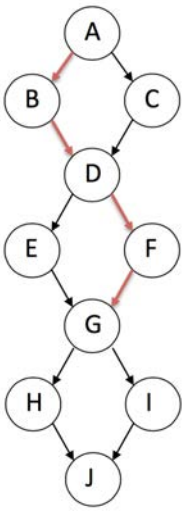


Figure 2.8 – Infeasible paths examples.

with $C_{l/i/r/j}$ non-negative integer constants and *op* an operator in set $\{=, >, \geq, <, \leq\}$.

2.2.2 Encoding

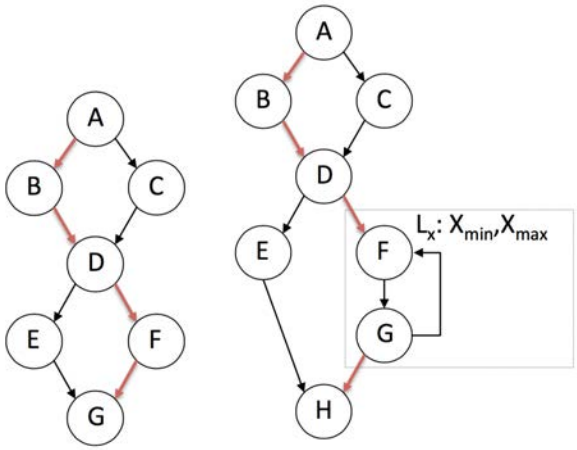
Note that loop bounds and infeasible paths, when finally used to calculate WCET using IPET, will eventually be encoded as linear constraints. For the translation of infeasibility into ILP constraints, there is a theoretical method proposed by Raymond in [Ray14].

For loop bounds, the steps of the encoding are:

- If the loop is a non nested loop, for each basic block i in the loop except the conditional test basic block, with the loop bounds LB_{min} and LB_{max}, we can derive the constraints: $f_i \geq LB_{min}$ and $f_i \leq LB_{max}$. For the test basic block, in the bottom tested loop, its constraint is the same as the other basic blocks. And in the top tested loop, its constraint is: $f_i \geq LB_{min} + 1$ and $f_i \leq LB_{max} + 1$.

- In the case of nested loops, the basic blocks in the outer loop body without inner loop are the same as no nested loop. The basic blocks in the inner loop body can be handled with the same method as no nested loop, only with the loop bounds $LB\_inner_{min} \times LB\_outer_{min}$ and $LB\_inner_{max} \times LB\_outer_{max}$. However, as illustrated later in the thesis, keeping the notion of loops is a richer information, and it integrates more naturally in a compiler.

And for infeasible paths, the encoding is a bit complex. It can be classified into the following two categories shown in Figure 2.9. The red paths in the two sugfigures are their corresponding infeasible paths.



(a) Case One (b) Case Two

Figure 2.9 – The CFG of infeasible paths examples.

- Case one shown in Figure 2.9a: The basic and general case. For this kind of infeasible path, the derived formal constraint is:

$$\sum f_i \leq m \times n \quad (i \in infeasible\_path)$$

m is the number of the basic blocks in the infeasible path minus one. n is a parameter. When this infeasible path is not in a loop or not in a function called multiple times, $n = 1$, otherwise, n should be the loop bound or the number of function called.

The constraint of case one is:

$$f_A + f_B + f_D + f_F + f_G \leq 4 \times n$$

In this case: $m = 4 = 5 - 1$).

- Case two shown in Figure 2.9b: In this case, there is a loop in the branch.

$$\sum f_i + \sum \frac{f_j}{X_{max}} \leq m \times n$$
$$(i \in infeasible\_path \ \& \ i \notin loop\_in\_infeasible\_path)$$
$$(j \in loop\_in\_infeasible\_path)$$

m is the number of the basic blocks in the infeasible path minus one. n is a parameter as the same as the previous case.

Considering that $\frac{f_j}{X_{max}}$ is not integral, the constraint should be transformed to:

$$\sum f_i \times X_{max} + \sum f_j \leq m \times n \times X_{max}$$

So the constraint of case two is:

$$f_A + f_B + f_D + \frac{f_F}{X_{max}} + \frac{f_G}{X_{max}} + f_H \leq 5 \times n$$

In this case: $m = 5 = 6 - 1$.

In this constraint, $\frac{f_F}{X_{max}}$ and $\frac{f_G}{X_{max}}$ is not integral, we transform it to:

$$(f_A + f_B + f_D + f_H) \times X_{max} + f_F + f_G \leq 5 \times n \times X_{max}$$

In both categories, when n is not 1, there is loss of information. However the constraints are still safe and can tighten WCET estimates. For example, the case shown in Figure 2.9a, the infeasible path is $A - B - D - F - G$. When $n = 1$, we can derive that the execution count of basic block B and F can be 1 at the same time. However, when n is not 1, *e.g.* $n = 10$, let us consider the following situation: the path $A - B - D - F - G$ is executed twice; the path $A - B - D - E - G$ is executed 6 times; the path $A - C - D - E - G$ is executed twice. This situation cannot happen, because the path $A - B - D - F - G$ is infeasible and should not be executed. But this situation can match our constraint. So information is lost when n is not 1.

2.3 Constraint Transformation Rules

The linear constraints represent the flow information in the original CFG corresponding to the source code. And they should be used with IPET technique to calculate the WCET result. However, the mapping between these constraints and the CFG/-code is broken by compiler optimizations. So, for optimizing compilation, these linear constraints are useless, even unsafe for the optimized binary code.

Our transformation framework is proposed to handle this problem. We analyze the most common compiler optimizations, According to their modification to the CFG, we can summarize a set of rules to rewrite the linear constraints. After modified with the transformation rules, the new linear constraints can match the CFG of the optimized binary code.

There are three basic rewriting rules for transforming flow information: *change rule*, *removal rule* and *addition rule*.

For each compiler optimization, a set of associated transformation rules (change, removal, addition) are defined in agreement to the CFG modifications. When the optimization is called, the corresponding rules are applied to transform flow information accordingly. Figure 2.10 shows an example: loop unrolling. The loop body is replicated twice. IR (Intermediate Representation) is a representation of the program used in compilers. When the IR is sent to the compiler, the corresponding CFG is modified by compiler optimizations. Our transformation framework defines transformation rule sets for these optimizations. In this way, the original flow information can be updated into the new one. In the figure, the loop bounds and the inequality of the relation of basic blocks are updated with the transformation rule set. After the optimization, a modified IR, a modified CFG and its corresponding new flow information are derived.

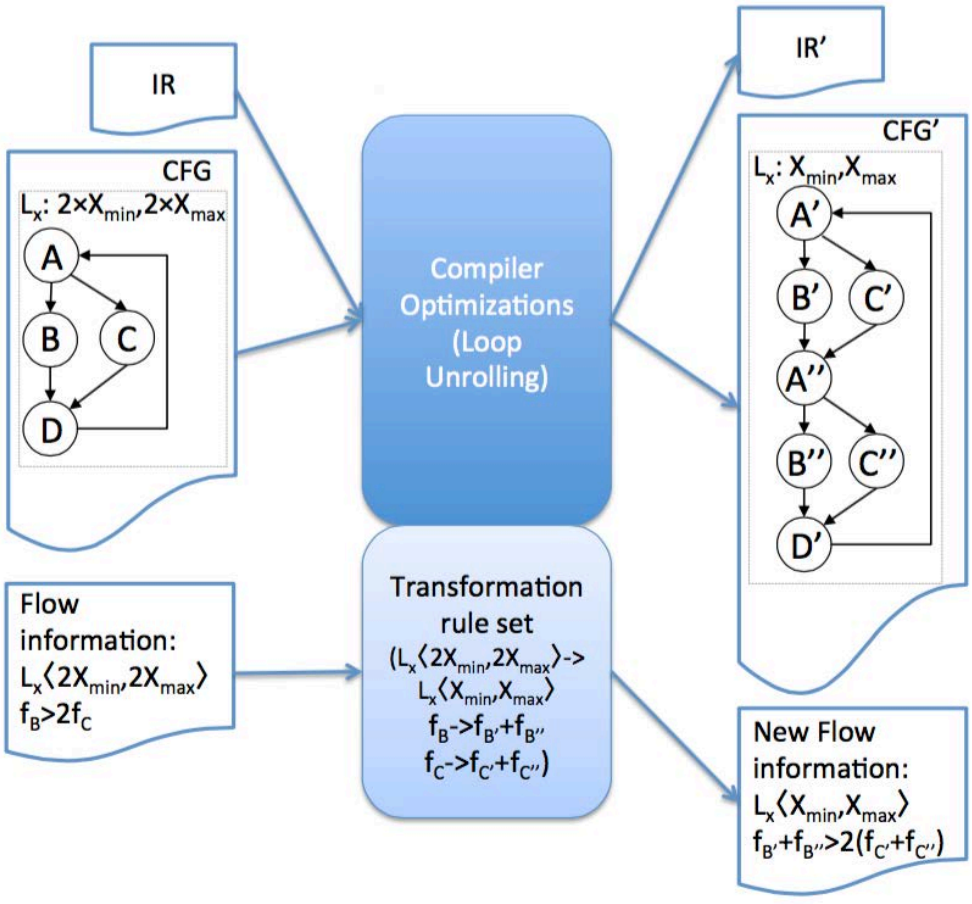


Figure 2.10 – Example of input and output of compiler optimization and transformation rule set

2.3.1 Change rule

This class of rule is used when the compiler optimization changes the execution count of basic blocks, or changes loop bounds. When the optimizations substitute β for α, we express it as $\alpha \rightarrow \beta$.

This rule contains two cases:

One case is the change of the execution count of a basic block. In this case, α is f_i, with i one of the basic blocks in the original CFG. β is then an expression $\{C_j + \sum_{j \in newCFG} M_j \times f_j\}$. In this expression, C_j is a constant and M_j is a multiplicative coefficient, that can be either a non-negative integer constant, an interval [a,b] or an interval [a,+∞) with both a and b non-negative constants.

Here, we use the CFG of two loops after the application of *loop spreading* as an example to demonstrate this class of rule. Loop spreading minimizes the parallel execution time by moving some computations from one loop to another. The modifications of source code and the CFG are shown in Figure 2.11. Figure 2.11a and Figure 2.11b give the source code before and after loop spreading, whereas Figure 2.11c and Figure 2.11d show their corresponding CFG. In Figure 2.11c, basic blocks C and D are the branches in the loop. Assuming that more than two-thirds of the elements in array c are greater than zero, then an additional constraint is needed to be added: $f_C \geq 2 \times f_D$, which means that the execution count of C is always no less than twice the execution count of D. This additional constraint is decided by the content of array c which is known by the programmers or through static analysis tools, but difficult to be derived by the compiler. The two loops have different loop bounds and have a dependence (array **a** is written by the first loop and read by the second loop), so they cannot be fused directly. Loop spreading is needed to move some iterations of loop body of L_y to L_x. The optimization divides node C into C' and C'', and same to D. So the rules should be $f_C \rightarrow f_{C'} + f_{C''}$ and $f_D \rightarrow f_{D'} + f_{D''}$. With the original constraint and rules, we can derive the new constraint $f_{C'} + f_{C''} \geq 2 \times (f_{D'} + f_{D''})$.

Another case is the change of a loop bound caused by a compiler optimization. α is a loop bound constraint $L_x \langle l_{bound}, u_{bound} \rangle$, with $L_x \subset original\_CFG$. β is $L_{x'} \langle l_{bound'}, u_{bound'} \rangle$ with $L_{x'} \subset new\_CFG$ and $l_{bound'}$ and $u_{bound'}$ new loop bounds which should be non-negative integer constants or any expression resulting a non-negative integer.

We still take the loop spreading as the example. In Figure 2.11, the loop bound of L_y is reduced from $X + Y$ to Y, so the transformation rule for loop bound should be $L_y \langle X_{min} + Y_{min}, X_{max} + Y_{max} \rangle \rightarrow L_y \langle Y_{min}, Y_{max} \rangle$. With this updated loop bound, we can derive that the execution count of all new basic blocks in this new loop should be no greater than Y_{max}.

2.3.2 Removal rule

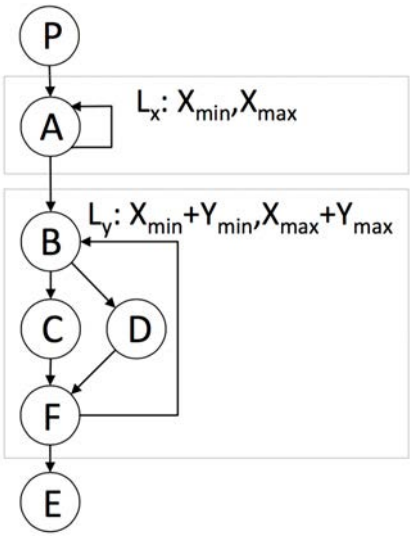
This class of rule is used whenever a basic block or a loop is removed from the CFG due to some compiler optimizations. We express it as $\alpha \rightarrow \emptyset$. α can be f_i ($i \in$ original_CFG) or $L_x \langle l_{bound}, u_{bound} \rangle$ ($L_x \subset$ original_CFG) depending on the object (basic block, loop)

```
for(i=0; i<X; i++)
{
  a[i]=a[i]+d;
}
for(i=0; i<X+Y; i++)
{
  if(c[i]>0)
    b[i]=a[i]+e;
  else
    b[i]=a[i]-e;
}
```

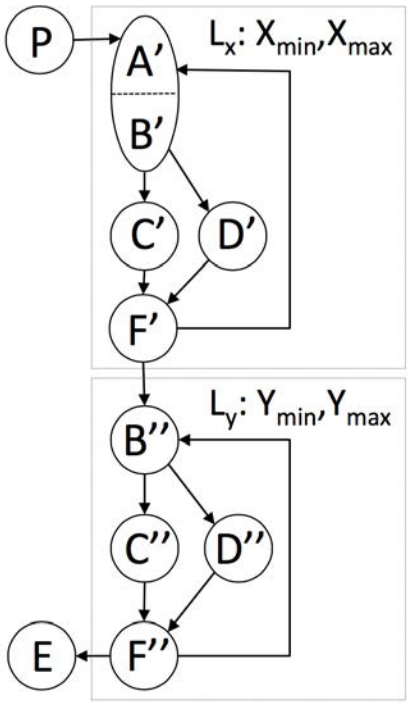
(a) Source code

```
for(i=0; i<X; i++)
{
  a[i]=a[i]+d;
  if(c[i]>0)
    b[i]=a[i]+e;
  else
    b[i]=a[i]-e;
}
for(i=X; i<X+Y; i++)
{
  if(c[i]>0)
    b[i]=a[i]+e;
  else
    b[i]=a[i]-e;
}
```

(b) Code after loop spreading



(c) Original CFG



(d) CFG after spreading

Figure 2.11 – The CFG of loop spreading example.

that is removed. Through this transformation, α is deleted from the constraints.

This rule is usually used when the elimination optimizations are applied, *e.g.* unreachable code elimination, dead code elimination and so on. These optimizations remove some unreachable or dead basic blocks or loops from the CFG.

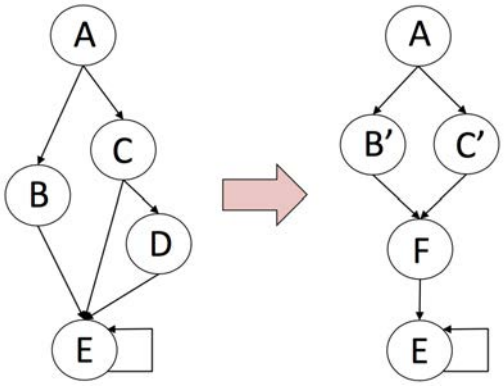


Figure 2.12 – The CFG of combination of unreachable-code elimination and tail merging example.

We can use the example in Figure 2.12 to illustrate this rule. The modifications of the CFG are shown in this figure. In this case, basic block D is an unreachable basic block. So the optimization *unreachable-code elimination* removes it from the CFG, and a removal rule is needed: $f_D \rightarrow \emptyset$. Assuming the initial additional constraint is $f_B \geq 3 \times f_D$, which means that the execution count of B is always no less than three times the one of D, this constraint should be deleted because of the remove of basic block D.

2.3.3 Addition rule

This class of rule is used when any new objects (basic block or loop) are added to the CFG by an optimization. When a new term is introduced into the CFG, the new constraint is added directly. The constraint should be linear, and should only involve objects (basic blocks, loops) from the new CFG.

For example, in Figure 2.12, after the removal of basic block D, because the instructions at the end of basic blocks B and C are the same and target same destination, these instructions are merged into a new basic block F which is introduced into the new CFG, so if there is any new constraint which involves the execution count of this new basic block with other basic blocks, we need to add it.

2.3.4 Rules manipulation

The parameter of the change rule has two forms: single value and interval. For the single value, *e.g.* $f_A \rightarrow f_B$, it is easy to apply by directly changing the value of the constraint term. While for the interval, *e.g.* $f_A \rightarrow [n_1 \dots n_2] f_B$, we need consider the relation used by the constraints where the replacement is made:

Relation "$>$" or "$\geq$" If the term which needs to be replaced occurs on the left of the relation, the upper bound of the interval is used in the new term. For the one on the right of the relation, we use the lower bound correspondingly. For example, we have an update rule $f_A \to [3 \dots 5] f_C$. So we need transform constraint $2f_A > 3f_B$ into $10f_C > 3f_B$, while in contrast, transform the constraint $3f_B > 2f_A$ into $3f_B > 6f_C$. And there is a special case: the interval is [a,$+\infty$), *i.e.* the rule is $\alpha \to [a, +\infty)$. In this case, if α occurs on the left of the relation, we should eliminate this constraint, *e.g.* for $f_A \to [2, +\infty) f_C$ and $2f_A > 3f_B$, $2f_A > 3f_B$ will be eliminated.

Relation "$<$" or "$\leq$" Converse of relation "$>$" or "$\geq$". For example, we have an update rule $f_A \to [3 \dots 5] f_C$. So we need transform the constraint $2f_A < 3f_B$ into $6f_C < 3f_B$, while in contrast, transform the constraint $3f_B < 2f_A$ into $3f_B < 10f_C$. There is also a special case: if the rule is $\alpha \to [a, +\infty)$, when α occurs on the right of the relation, we should eliminate this constraint.

Relation "$=$" When the term is in the "$=$" relation, it can't be updated with the interval transformation. We need apply a normalization before the transformation. For example, turn a constraint $f_A = f_B$ into $f_A \leq f_B$ and $f_A \geq f_B$. Then the two inequality relations semantics can be applied.

2.3.5 Operations after transformation

After the transformation rules are applied, the new constraints might not satisfy the standards setted in Section 2.2 (constants as multiple factors). For example, a main problem is that there are fractions in the constraints after the transformation. So we still need the following operations to standardize the new set of constraints. These operations should be executed in the following order:

Integralization Because some transformation rules contain fractions, the new obtained constraints may contain fractions. But fractions are not allowed in ILP. So we need the integralization for the conversion from fraction to integer by multiplying the denominator on both sides of the constraints. For example, with constraint $2f_A \leq 3f_B$ and rule $f_A \to \frac{4}{3} f_C$, we can get the new constraint $2 \times \frac{4}{3} f_C \leq 3f_B$. The integralization is needed because of the left side of the relation. The constraint becomes $2 \times 4f_C \leq 3 \times 3f_B$.

Simplification The transformation and integralization both bring two kinds of constraints which need to be simplified:

- The coefficients can be algebraic simplified. For example, constraint $2 \times 4f_C \leq 3 \times 3f_B$ is simplified to $8f_C \leq 9f_B$.
- The coefficients and constants on both sides have greatest common divisor, *e.g.* the constraint $9f_A + 6f_C \leq 12f_B + 3$ can be simplified to $3f_A + 2f_C \leq 4f_B + 1$.

Redundancy elimination This is the final operation of this framework. After the simplification, we traverse all constraints, and eliminate the redundancy. The redundancy includes the same constraints, also the constraints like: $2f_A \leq 3$ and $2f_A \leq 5$.

After applying the transformation rule set of each optimization, the new constraints are created. Then we do the integralization to these new constraints. Afterwards, we need simplify these constraints. At the end, combining with structural constraints, we check the redundancy and eliminate them.

For the implementation in the next chapter, we trace only loop bounds, and new loop bounds are stored as integers. So these operations are not needed in our implementation.

2.3.6 The influence of transformation framework on estimated WCET

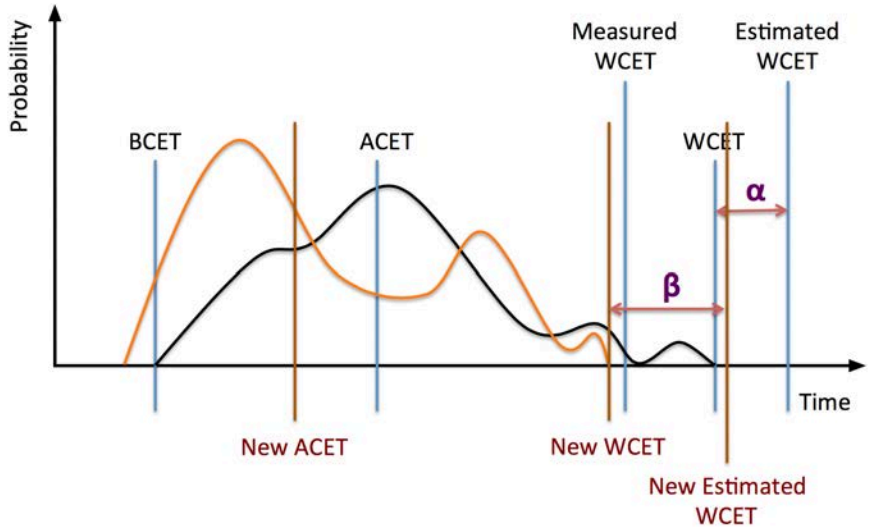


Figure 2.13 – Distribution of execution time before and after compiler optimizations

These transformation semantics keep the WCET calculation safe if transformation rules are expressed correctly, while information may be lost during the transformation. We use an example to demonstrate our viewpoint.

Let us assume that there is a constraint $2f_A > 3f_B$ and an update rule $f_A \rightarrow [3 \ldots 5]f_C$, we can get a new constraint $10f_C > 3f_B$. However now we can only promise that the transformation is safe, but maybe not precise. Because if the fact is $n \times f_C > 3f_B$ where $6 \leq n < 10$, with the new constraint, the information may be lost. When there is an interval in the transformation rule, our new constraint expresses the worst but safe case, but may be not the precise representation of new situation.

The curves in Figure 2.13 represent the probability of different execution times of a program with different input data and initial processor states. The black and right one depicts the execution time without optimizations, and the orange and left one

describes the optimized execution time. In general, after the optimizations, as the figure shows that the ACET is reduced because the compiler optimizations mostly aim at minimizing ACET. Many researches [DAfESG, FLT06, FS06] prove that WCET can also benefit from compiler optimizations. So in the figure, WCET also decreases after the optimizations. The new estimated WCET is the WCET result produced with our transformation framework. The experimental result in Chapter 4 demonstrates that the new estimated WCET is reduced compared with the original one. However, when we consider α (the difference between the original actual WCET and estimated WCET, considered as precision) and β (the difference between the optimized actual WCET and estimated WCET), we cannot compare them, because the actual WCET is in general unknown. So we cannot verify that our transformation framework can improve the precision, *i.e.* we cannot prove that β is smaller than α.

Through inducing the transformation rule sets of different compiler optimizations and analyzing their effect on the flow information, we can draw the following three conclusions:

- Our transformation framework is safe, *i.e.* $\beta > 0$ which is important for hard real-time systems.
- If the execution count of nodes and loop bounds can be derived, with our transformation framework, no flow information is lost for almost all optimizations, and the new estimated WCET is better (*i.e.* smaller) than the original one.
- Flow information can be transformed correctly with our transformation framework, and the experimental results in Chapter 4 prove this.

2.4 Overview of Transformation Framework

Figure 2.14 illustrates the overview of our transformation framework within the compilation and WCET estimation. The right column is the flow of standard code compilation and WCET estimation. Without optimizations, the source level flow information can be used directly at binary level. When the optimizations are applied, we need a transformation framework to transform source-level annotations to binary-level annotations. Our transformation framework is at the same level as compilers. What is crucial is that transforming the flow information is done *within* the compiler, in parallel with transforming the code. With these binary-level annotations, WCET analysis tools can calculate estimated WCET result.

2.5 Compiler Optimizations and Their Concrete Transformation Rules

Modern compilers can have a huge effect on code performance. This benefits from the optimizations included in the compilers. These optimizations improve the generated

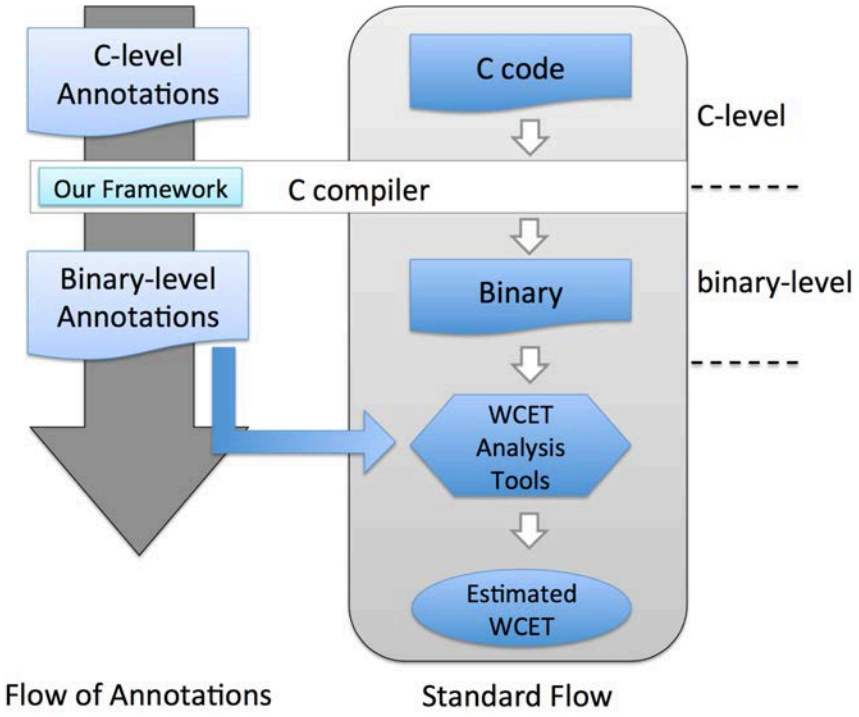


Figure 2.14 – Overall flow

code in different ways while ensuring the result of the code is identical. In general, optimizations application is decided by the following two fundamental criteria:

Speed: improving the runtime performance of the generated code

Space: reducing the size of the generated code

In most cases, maximizing speed is more important than minimizing space. Ideally, we expect that the optimizations can achieve both criteria. Unfortunately, many "speed" optimizations make the code larger, and many "space" optimizations increase execution time – this is known as the space-time tradeoff.

In this section, we begin the presentation of a series of optimizations[6] which focus on different phases, different types of code. Then according to the transformation framework, we analyze these optimizations and describe their transformation rule sets. Here, we just list the optimizations analyzed in our transformation framework.

2.5.1 Redundancy elimination, procedure, control-flow and low-level optimizations

This part contains the most general optimizations except loop optimizations and vectorization optimizations. For example, redundancy elimination optimizations deal with

[6]The optimizations in this section are the ones we support.

the identification and elimination of redundant computations; procedure optimizations are the compiler techniques which improve performance in programs containing many frequently used functions of small or medium size. It differs from other compiler optimizations because it analyzes the entire program instead of only a single function, or even a single block of code. Besides, this subsection introduces some control-flow and low-level optimizations.

2.5.1.1 Unreachable code elimination

Unreachable code is part of the source code of a program which can never be executed, regardless of the input data. There never exists control flow path to this code from the rest of the program.

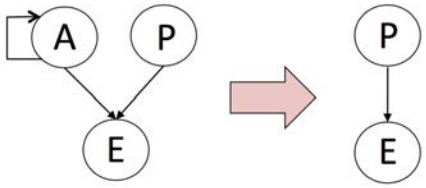
This elimination has no direct effect on the execution speed, but decreases unnecessary memory the program occupies. Thus may have secondary effects on the speed. Also, the elimination may enable other further optimizations. Figure 2.15 shows a simple example.

```
int foo(int a, int b){
  c = a - b;
  goto ret;
  while(1){
    c++;
  }
ret:
  return c;
}
```

(a) Original source code

```
int foo(int a, int b){
  c = a - b;
  return c;
}
```

(b) Optimized code



(c) CFG of unreachable code elimination

Figure 2.15 – Unreachable code elimination example

The transformation rule set[7] for unreachable code elimination is:

$$f_{BB\_unreachable} \rightarrow \emptyset \tag{2.1}$$

For the example of CFG in Figure 2.15c, the transformation rules for unreachable basic block can be described:

[7]The BB in the transformation rule represents Basic Block, and is used in the rest of this section.

$$f_A \rightarrow \emptyset \tag{2.2}$$

For the rule set of this optimization, no flow information is lost.

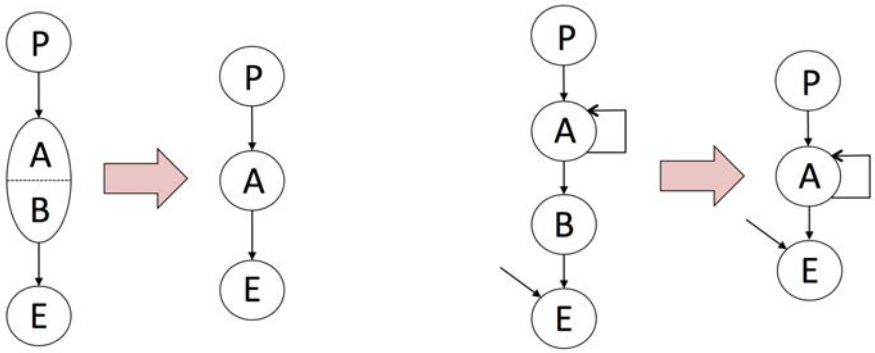
2.5.1.2 Dead code elimination

```
int foo(void){
    int a = 16;
    int b = 18;
    int c = a << 2;
    int d = b << 3;
    return c;
}
```

(a) Original source code

```
int foo(void){
    int a = 16;
    int c = a << 2;
    return c;
}
```

(b) Optimized code



(c) Elimination of dead code in Basic Block (d) Elimination of entire Basic Block

Figure 2.16 – Dead code elimination example

Dead code elimination is similar to unreachable code elimination, the difference is the dead code is executed but has no effect on the result. Dead code elimination removes this dead code. Dead code may already exist in source code, but it is more likely arisen from optimizations. By removing dead code, *Dead code elimination* can reduce code size and improve performance. A simple example is shown in Figure 2.16.

The transformation rule set for dead code elimination should consider two cases: For the case in Figure 2.16c, we just update the node:

$$f_{BB\_withdeadcode} \rightarrow f_{BB\_withoutdeadcode} \tag{2.3}$$

For the case in Figure 2.16d, we need the removal rule:

$$f_{BB\_deadcode} \rightarrow \emptyset \tag{2.4}$$

So for the example of CFG in Figure 2.16, the corresponding transformation rules for deleting dead code are:

Case of elimination of dead code in Basic Block:

$$f_{AB} \rightarrow f_A \tag{2.5}$$

Case of elimination of entire Basic Block:

$$f_B \rightarrow \emptyset \tag{2.6}$$

For the rule set of this optimization, no flow information is lost.

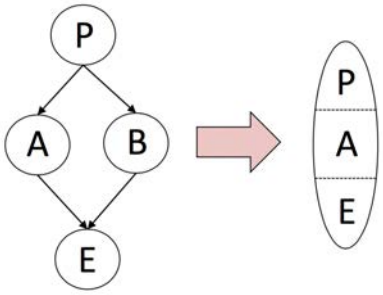
2.5.1.3 If simplification

```
if(a>0){
   b=a;
   c=b*b;
   if(b>0)
      d=2;
   else
      d=0;
}
```

(a) Original source code

```
if(a>0){
   b=a;
   c=b*b;
   d=2;
}
```

(b) Optimized code



(c) CFG of if simplification

Figure 2.17 – If simplification example

If simplification is applied to a conditional construct in which there are branches never taken. This optimization removes the empty or the not taken branch, and it can reduce the overhead of conditional branching. An example is shown in Figure 2.17. This optimization can remove part of the CFG and simplify the flow information.

The transformation rule set for if simplification is:

$$f_{BB\_branchnotaken} \rightarrow \emptyset \tag{2.7}$$

By using the general rule set, we can get the following set of transformation rules from the CFG in Figure 2.17c:

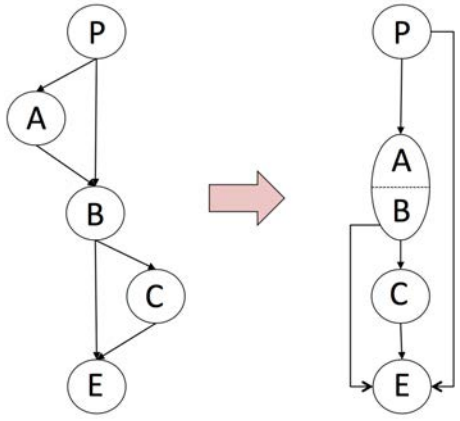
$$\begin{aligned} f_P &\rightarrow f_{PAE} \\ f_A &\rightarrow f_{PAE} \\ f_E &\rightarrow f_{PAE} \\ f_B &\rightarrow \emptyset \end{aligned} \tag{2.8}$$

The first three rules are created because of the removal of node B.
Although these update rules change the CFG, no flow information is lost.

2.5.1.4 Branch optimization

```
    if(a=0)                    if(a=0)
      goto L1;                   goto L2;
    . . .                      . . .
L1: if(a<5)                L1: if(a<5)
      goto L2;                   goto L2;
    . . .                      . . .
L2: . . .                  L2: . . .
```

(a) Original source code | (b) Optimized code



(c) CFG of branch optimization

Figure 2.18 – Branch optimization example

If there are more than one successive branches, we need branch optimization. The general principle of this optimization is to eliminate the unconditional branches. If there are two successive conditional branches and the two conditions have relation, they can be rewritten with the test of the first branch to the target of the second branch (example in Figure 2.18).

Branch optimization can reduce the number of conditional or unconditional branches. A sample example of *branch optimization* is shown in Figure 2.18.

In general, this optimization just modifies the control flow, for unconditional branches, no node is modified. And for the two successive conditional branches, the execution count of the second branch test node is changed.

When the compiler knows the execution count of the nodes of first branch test ft and branch fb:

$$f_{second\_test} \rightarrow \frac{f_{ft}}{f_{fb}} f_{second\_test'} \tag{2.9}$$

Else:

$$f_{second\_test} \rightarrow [1 \ldots \infty] f_{second\_test'} \tag{2.10}$$

As the example shown in Figure 2.18, *branch optimization* skips the jump to B and directly goes to B' successor E. With these information, we can describe the following rule:

When the compiler knows the execution count of node P and A:

$$f_B \rightarrow \frac{f_P}{f_A} f_{AB} \tag{2.11}$$

Else:

$$f_B \rightarrow [1 \ldots \infty] f_{AB} \tag{2.12}$$

In the first case, no flow information is lost. While in the second case, the new execution count of node AB cannot be calculated precisely, because of the interval in the rule.

2.5.1.5 Tail merging (cross jumping)

Tail merging, also known as *cross jumping*, is an optimization eliminating duplicates in the code. It is identifying sequences of identical instructions at the end of different basic blocks targeting same destination and transforms these instructions into a new basic block. This new basic block is shared by those different basic blocks. This optimization always saves code space and may also save time. A sample example is shown in Figure 2.19.

The general transformation rule set of tail merging is (A represents the different basic blocks that targets the same destination, and the codes in A change after the optimizations. These code changes are denoted by a prime following the basic blocks names, and this is used in the rest of thesis.):

$$\begin{aligned} f_{BB\_A} &\rightarrow f_{BB\_A'} \\ f_{BB\_target} &\rightarrow f_{BB\_target'} \end{aligned} \tag{2.13}$$

The code transformation of *Tail Merging* is given by Figure 2.19c. With this information, the following update rules are induced:

$$\begin{aligned} f_B &\rightarrow f_{B'} \\ f_C &\rightarrow f_{C'} \\ f_E &\rightarrow f_{DE} \end{aligned} \tag{2.14}$$

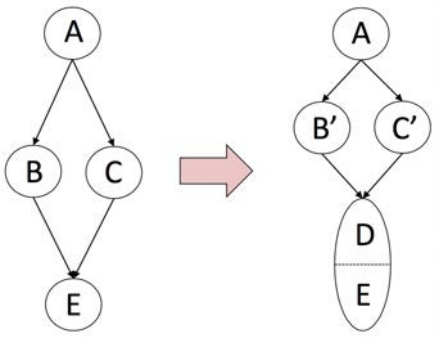
For the rule set of this optimization, no flow information is lost.

```
if (a<0){
   codeA;
   codeC;
}
else{
   codeB;
   codeC;
}
```

(a) Original source code

```
if (a<0){
   codeA;
}
else{
   codeB;
}
codeC;
```

(b) Optimized code



(c) CFG of tail merging

Figure 2.19 – Tail merging example

2.5.1.6 Inlining

Inlining is an optimization that replaces a function call site with the body of the called function. It moves the code from the called functions to local code, which not only improves its effects, but that can be optimized as part of calling code, as shown in Figure 2.20 *constant propagation* can be applied to improve the code. But the disadvantage is obvious: the code size is increased when the function is called many times.

```
inline int f(int n){
  n = n - 5;
  return n;
}

int x=3;
int y=f(x);
printf("%d",y);
```
(a) Original source code

```
int x=3;
x=x-5;
int y=x;
printf("%d",y);
```
(b) Optimized code

Figure 2.20 – Inline

For inlining, we need to move the flow information constraints from functions to the called function.

2.5.2 Loop optimizations

Optimizations that apply to loops are essential to achieving high performance. The optimizations covered in this subsection operate on loops, and they are described in the following.

2.5.2.1 Loop unrolling

Loop unrolling reschedules the loop and replicates the body of the loop in one iteration according to the unrolling factor. It can reduce loop overhead and increase instruction-level parallelism. Loop unrolling replicates the loop body UF times (UF stands for unrolling factor). There are two basic types of *loop unrolling* optimizations: lucky case: loop count is constant and a multiple of unrolling factor (see Figure 2.21); and general case: loop count is not a multiple of unrolling count (shown in Figure 2.22).

The modifications of the CFG are shown in Figure 2.23, in the general case where the loop bound is not known to be a multiple of the unrolling factor. In the figure, the loop body A is replicated UF times and the structure of the CFG is changed; a new loop is created to cope with number of iterations not multiple of UF. The loop bound of these two loops are also different from the original one.

The transformation of flow information for loop unrolling requires the application of the change rule and the addition rule because of the addition of the new loop. The set of rules describing the flow transformation is given below (new_A represents the node containing $A_1 \dots A_{UF}$).

`for(i=0; i<10; i++)`
`  a[i]=a[i]+b[i];`
	`(unrolling factor=2)`
	`for(i=0; i<10; i+=2){`
	`  a[i]=a[i]+b[i];`
	`  a[i+1]=a[i+1]+b[i+1];`
	`}`

(a) Original source code	(b) Optimized code

Figure 2.21 – Loop unrolling example (loop_count%UF = 0)

`for(i=0; i<10; i++)`
`  a[i]=a[i]+b[i];`
	`(unrolling count=3)`
	`for(i=0; i<floor(10/3)*3; i+=3){`
	`  a[i]=a[i]+b[i];`
	`  a[i+1]=a[i+1]+b[i+1];`
	`  a[i+2]=a[i+2]+b[i+2];`
	`}`
	`a[i]=a[i]+b[i];`

(a) Original source code	(b) Optimized code

Figure 2.22 – Loop unrolling example (loop_count%UF != 0)

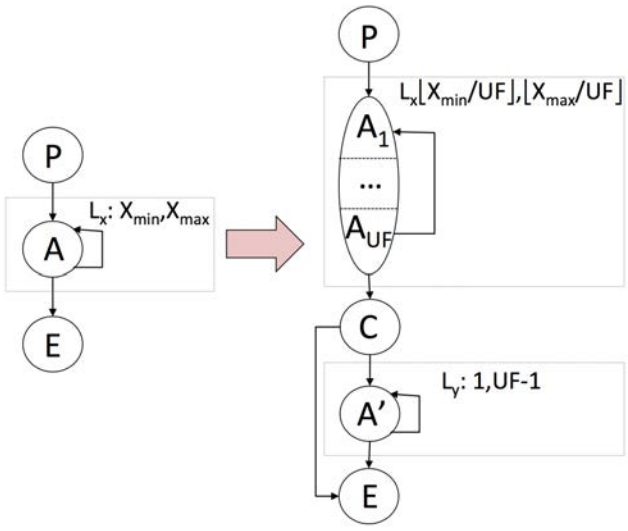


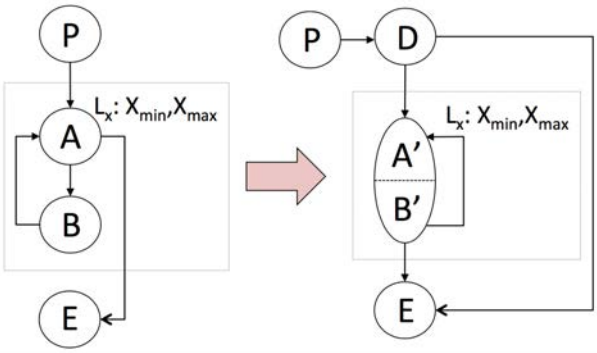
Figure 2.23 – CFG of loop unrolling example. The left part of the figure shows the original CFG, whereas the right part shows the optimized one.

```
while(i<n){
  a[i]=i;
  i++;
}
```

(a) Original source code

```
if(i<n){
  do{
    a[i]=i;
    i++;
  }while(i<n);
}
```

(b) Optimized code



(c) CFG of loop rotation

Figure 2.24 – Loop rotation example

$$\begin{aligned} L_X \langle X_{\min}, X_{\max} \rangle \rightarrow & L_X \left\langle \lfloor \frac{X_{\min}}{UF} \rfloor, \lfloor \frac{X_{\max}}{UF} \rfloor \right\rangle \\ & L_Y \langle 1, UF - 1 \rangle \\ f_A \rightarrow & UF \times f_{new\_A} + f_{A'} \end{aligned} \tag{2.15}$$

The first line (change rule) expresses that the loop bound of the first loop is derived from the loop bound of the original loop by dividing it by the unrolling factor UF. The second line (addition rule) expresses the loop bound of the new loop. The following line (change rule) updates the constraints on the basic block to reflect the alteration of its execution count.

Although the basic blocks and loops are modified, for the rule set of this optimization, no flow information is lost.

2.5.2.2 Loop inversion (loop rotation)

Loop inversion, also known as *loop rotation* is an optimization in which the loop with the test at the start of the loop is replaced with the test at the end of the loop. A simple example is shown in Figure 2.24.

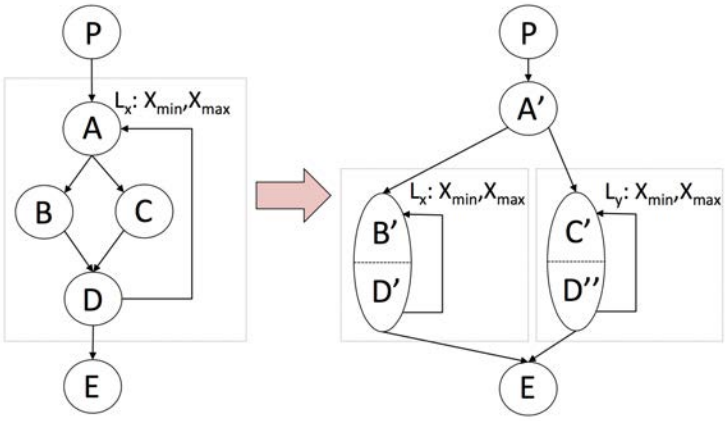
As shown in the Figure 2.24c, the transformation rules of *loop rotation* are described:

```
for(i=0; i<n; i++){
  a[i]=b[i];
  if(x>0)
    a[i]=a[i]+1;
  else
    a[i]=a[i]*2;
}
```

(a) Original source code

```
if(x>0)
  for(i=0; i<n; i++){
    a[i]=b[i];
    a[i]=a[i]+1;
  }
else
  for(i=0; i<n; i++){
    a[i]=b[i];
    a[i]=a[i]*2;
  }
```

(b) Optimized code



(c) CFG of loop unswitch

Figure 2.25 – Loop unswitch example

$$\begin{aligned} f_A &\rightarrow f_{A'B'} + f_D \\ f_B &\rightarrow f_{A'B'} \\ f_D &= 1 \end{aligned} \tag{2.16}$$

For the rule set of this optimization, no flow information is lost.

2.5.2.3 Loop unswitch

Loop unswitch moves a loop invariant test condition outside of the loop and duplicates the loop body inside each conditional branch. In this way, it can save the overhead of conditional branching inside the loop, reduce the loop code size and improve locality. An example of this optimization is shown in Figure 2.25.

The Figure 2.25c is the transformation of *loop unswitch*, the following set of rules can be described:

$$\begin{aligned} L_X\left\langle X_{\min}, X_{\max}\right\rangle \rightarrow & L_X\left\langle X_{\min}, X_{\max}\right\rangle, L_Y\left\langle X_{\min}, X_{\max}\right\rangle \\ f_B \rightarrow & f_{B'D'} \\ f_C \rightarrow & f_{C'D''} \\ f_D \rightarrow & f_{B'D'} + f_{C'D''} \end{aligned} \tag{2.17}$$

For this transformation rule set, no flow information is lost.

2.5.2.4 Loop deletion

Loop deletion is applied when the loop is demonstrated to never execute or the result of the loop has no effect on the computation of the function's final return value. *Loop deletion* can reduce code size and improve performance.

The transformation rule set is:

$$\begin{aligned} L_X\left\langle X_{\min}, X_{\max}\right\rangle \rightarrow & \emptyset \\ f_{BB\_inloop} \rightarrow & \emptyset \end{aligned} \tag{2.18}$$

2.5.2.5 Loop interchange

Loop interchange exchanges the order of two loops in a perfect loop nest. In general, it switches the outer loop to the inner position and vice versa. This optimization can improve vectorization and parallel performance, and reduce stride. In Figure 2.26, the original inner loop accesses the array a with stride m, after the optimization, it becomes stride 1 access and this is good for locality.

Loop interchange is typically applied to ensure that elements of a multi-dimensional array are accessed in the order in which they are represented in memory, improving locality of reference and thus performance in architectures with data caches.

The modifications of the CFG due to loop interchange are shown in Figure 2.26c. In the figure, it is assumed that the structure of the CFG (structure of loops, basic blocks) is not altered, although the contents of individual basic blocks due to loop interchange may change.

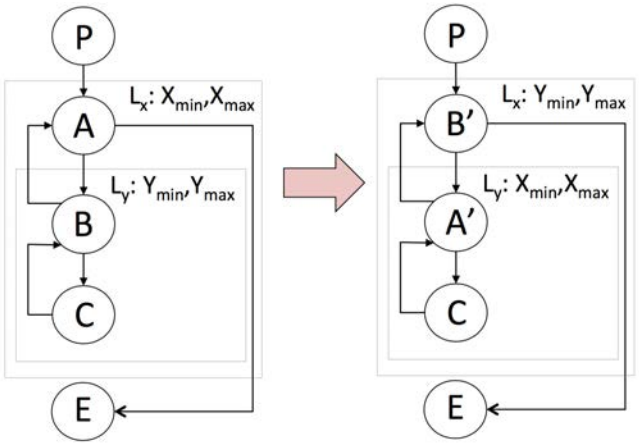
The transformation of flow information for the loop interchange optimization only requires the application of the change rule since there is no addition or removal of nodes and basic blocks. The set of rules describing the flow transformation is given below.

$$\begin{aligned} L_X\left\langle X_{\min}, X_{\max}\right\rangle \rightarrow & L_X\left\langle Y_{\min}, Y_{\max}\right\rangle \\ L_Y\left\langle Y_{\min}, Y_{\max}\right\rangle \rightarrow & L_Y\left\langle X_{\min}, X_{\max}\right\rangle \\ f_A \rightarrow & [\frac{1}{Y_{\max}} \cdots \frac{1}{Y_{\min}}] f_{A'} \\ f_B \rightarrow & [X_{\min} \ldots X_{\max}] f_{B'} \\ f_C \rightarrow & f_{C'} \end{aligned} \tag{2.19}$$

The first two lines show that the respective loop bounds of the two loops have been swapped. The following three lines update the constraints of the basic blocks to reflect

```
for(i=0; i<m; i++)                for(j=0; j<n; j++)
  for(j=0; j<n; j++)                for(i=0; i<m; i++)
    a[j,i]=a[j,i]+1;                  a[j,i]=a[j,i]+1;
```

(a) Original source code | (b) Optimized code



(c) CFG of loop interchange

Figure 2.26 – Loop interchange example

the alteration of their execution count. For example, the execution count f_B of node B changes from $[X_{\min} \dots X_{\max}] \times [Y_{\min} \dots Y_{\max}]$ to $[Y_{\min} \dots Y_{\max}]$, so the original f_B is replaced by the $[X_{\min} \dots X_{\max}]f_B$. If $Y_{\min}$ is 0, we should use $+\infty$ instead of $\frac{1}{Y_{\min}}$.

Considering the general situation, the rule set is safe but flow information is lost. However, when the exact loop bounds are known, no flow information is lost.

2.5.2.6 Loop distribution

Loop distribution also known as *loop fission* or *loop splitting* breaks a loop into multiple loops with the same iteration space as the original and a subset of the original loop body. It can be used to create perfect loop nest or loops with fewer dependences, and improve locality and register reuse. The Figure 2.27 shows an example of *loop distribution*. In this example, through *loop distribution*, a perfect loop nest is created, and then we can apply *loop interchange* to reduce the stride.

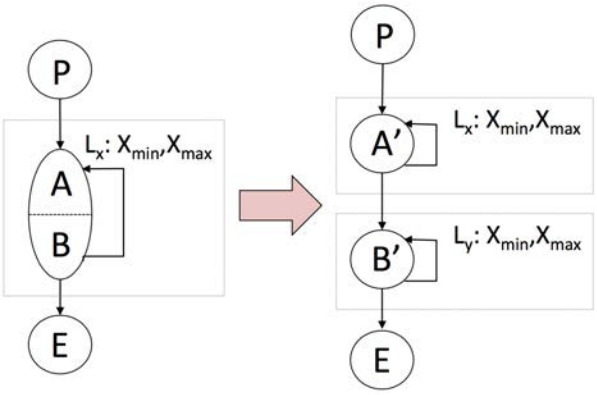
The code transformation of *loop distribution* is given by Figure 2.27c. The two new loops have the same loop bound with the original one. With this information, the following update rules are described:

```
for(i=0; i<n; i++){
   a[i]=b[i];
   for(j=0; j<m; j++){
      c[j,i]=d[j,i]+1;
   }
}
```

(a) Original source code

```
for(i=0; i<n; i++){
   a[i]=b[i];
}
for(i=0; i<n; i++){
   for(j=0; j<m; j++){
      c[j,i]=d[j,i]+1;
   }
}
```

(b) Optimized code

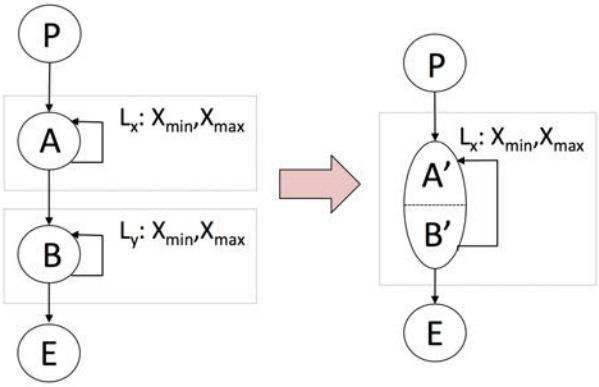


(c) CFG of loop distribution

Figure 2.27 – Loop distribution example

```
for(i=0; i<n; i++){
  b[i]=a[i]+3;
}
for(i=0; i<n; i++){
  c[i+1]=c[i]*2+b[i];
}
```
(a) Original source code

```
for(i=0; i<n; i++){
  b[i]=a[i]+3;
  c[i+1]=c[i]*2+b[i];
}
```
(b) Optimized code



(c) CFG of loop fusion

Figure 2.28 – Loop fusion example

$$\begin{aligned} L_X\left\langle X_{\min}, X_{\max}\right\rangle \rightarrow & L_X\left\langle X_{\min}, X_{\max}\right\rangle, L_Y\left\langle X_{\min}, X_{\max}\right\rangle \\ f_{AB} \rightarrow & \frac{1}{2} f_{A'} + \frac{1}{2} f_{B'} \\ f_{A'} = & f_{B'} \end{aligned} \tag{2.20}$$

For the rule set of this optimization, no flow information is lost.

2.5.2.7 Loop fusion

Loop fusion rewrites multiple loops with the same loop bound into a single one. It is an inverse optimization of loop distribution. An example is shown in Figure 2.28. After the fusion, $b[i]$ just needs to be loaded once in each iteration, thus improves register and cache locality.

The code transformation of *loop fusion* is given by Figure 2.28c. The two original loops must have the same loop bound, and the loop bound of the new loop is same with them. With this information, the following update rules are described:

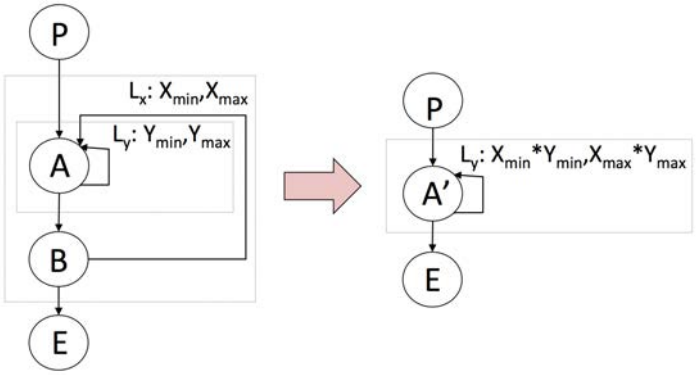
$$\begin{aligned} L_Y\left\langle X_{\min}, X_{\max}\right\rangle \rightarrow & L_X\left\langle X_{\min}, X_{\max}\right\rangle \\ f_A \rightarrow & f_{A'B'} \\ f_B \rightarrow & f_{A'B'} \end{aligned} \tag{2.21}$$

```
for(i=0; i<n; i++)
  for(j=0; j<m; j++)
    a[i,j]=a[i,j]+c;
```

(a) Original source code

```
for(p=0; p<n*m; p++){
  i=(p/m)*m;
  j=p mod m;
  a[i,j]=a[i,j]+c;
}
```

(b) Optimized code



(c) CFG of loop coalescing

Figure 2.29 – Loop coalescing example

For the rules of this optimization, no flow information is lost.

2.5.2.8 Loop coalescing

Loop coalescing turns a loop nest into a single loop. This optimization performs better on a parallel machine. It can improve the scheduling of the loop and reduce loop overhead. An example is shown in Figure 2.29. For a parallel machine, if the number of the processors P is slightly smaller than n and m, the first P iterations and the rest $m - P$ iterations take the same time. After the optimization, in addition to the last $n \times m \ mod \ P$ iterations, the first several P iterations can make full use of the P processors.

The code transformation of *loop coalescing* is given by Figure 2.29c. With this information, the following update rules are described:

$$\begin{aligned} L_X \langle X_{\min}, X_{\max} \rangle &\rightarrow L_Y \langle X_{\min} \times Y_{\min}, X_{\max} \times Y_{\max} \rangle \\ L_Y \langle X_{\min}, X_{\max} \rangle &\rightarrow L_Y \langle X_{\min} \times Y_{\min}, X_{\max} \times Y_{\max} \rangle \\ f_A &\rightarrow f_{A'} \end{aligned} \tag{2.22}$$

The rule set does not lose flow information.

2.5.2.9 Loop collapsing

Loop collapsing is similar to *loop coalescing*, while it is a less general version, because

`for(i=0; i<n; i++)`	`float * pa=a;`
`  for(j=0; j<m; j++)`	`for(p=0; p<n*m; p++)`
`  a[i,j]=a[i,j]+c;`	`  pa[p]=pa[p]+c;`
(a) Original source code	(b) Optimized code

Figure 2.30 – Loop collapsing example

it reduces the number of dimensions of the array. A simple example is shown in Figure 2.30. The advantage of this optimization is the overhead elimination of multi-dimensional array indexing.

The transformation of CFG due to this optimization is similar to loop coalescing. And the transformation rule set is also analogous.

2.5.2.10 Loop peeling

Loop peeling removes a small number of iterations from the start or end of the loop and lets these code be executed separately outside of the loop. It can enable parallelization and fusion. For the example in Figure 2.31, after this transformation, *loop fusion* can be applied.

The code transformation of *loop peeling* is given by Figure 2.31. With this information, the following update rules are described:

$$\begin{aligned} L_X\left\langle X_{\min}, X_{\max}\right\rangle &\rightarrow L_X\left\langle X_{\min}-k, X_{\max}-k\right\rangle \\ f_A &\rightarrow k \times f_{A_1 \ldots A_k}+f_{A'} \end{aligned} \tag{2.23}$$

For the rule set of this optimization, no flow information is lost.

2.5.2.11 Loop spreading

Loop spreading is designed to minimize the parallel execution time of a loop by moving some computation from one loop to another. In the example of Figure 2.32, the two loops have different loop bounds and have a dependence (array **a** is written by the first loop and read by the second loop), so they cannot be fused directly. In this case, we need the *loop spreading* to move some computation together.

The code transformation of *loop spreading* is given by Figure 2.32c. With this information, the following update rules are induced:

$$\begin{aligned} L_Y\left\langle Y_{\min}, Y_{\max}\right\rangle &\rightarrow L_Y\left\langle Y_{\min}-X_{\min}, Y_{\max}-X_{\max}\right\rangle \\ f_A &\rightarrow f_{A'B'} \\ f_B &\rightarrow f_{A'B'}+f_{B''} \end{aligned} \tag{2.24}$$

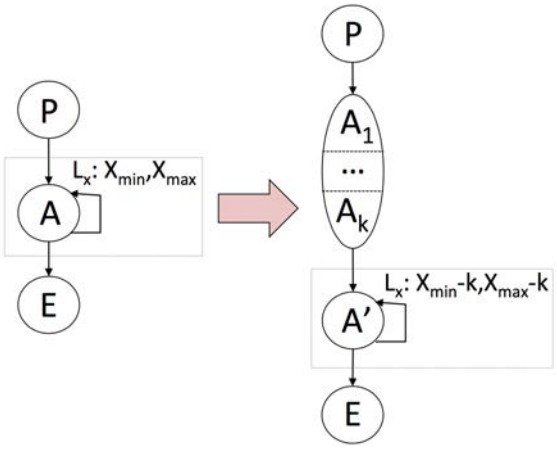
The rule set of this optimization does not lose flow information.

```
for(i=0; i<n; i++)
  a[i]=a[i]+c;
for(i=1; i<n; i++)
  b[i]=b[i]*2;
```

(a) Original source code

```
if(n>0)
  a[0]=a[0]+c;
for(i=1; i<n; i++)
  a[i]=a[i]+c;
for(i=1; i<n; i++)
  b[i]=b[i]*2;
```

(b) Optimized code



(c) CFG of loop peeling

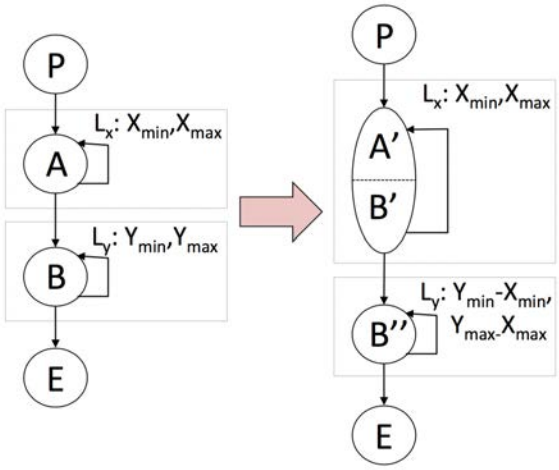
Figure 2.31 – Loop peeling example

```
for ( i =0;  i <n-m;  i++)
   a [ i ]= i +2;
for ( i =0;  i <n ;  i++)
   b [ i ]=b [ i ]+a [ i ] ;
```

(a) Original source code

```
for ( i =0;  i <n-m;  i++){
   a [ i ]= i +2;
   b [ i ]=b [ i ]+a [ i ] ;
}
for ( i=n-m;  i <n ;  i++)
   b [ i ]=b [ i ]+a [ i ] ;
```

(b) Optimized code

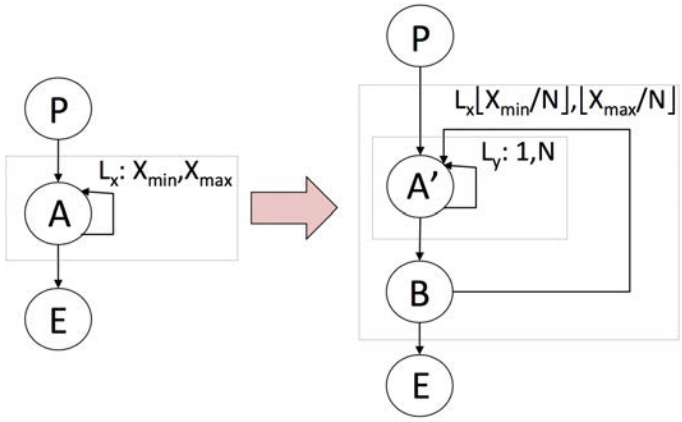


(c) CFG of loop spreading

Figure 2.32 – Loop spreading example

```
for(i=0; i<n; i++)
  a[i]=a[i]+c;
```
(a) Original source code

```
for(TI=0; TI<n; TI+=N)
  for(i=TI; i<min(TI+N,n); i++)
    a[i]=a[i]+c;
```
(b) Optimized code



(c) CFG of loop tiling

Figure 2.33 – Loop tiling example

2.5.2.12 Loop tiling (loop blocking)

Loop blocking, also known as *loop tiling* divides the iteration space of a loop into tiles or blocks, in this way it can enhance cache reuse. An example is shown in Figure 2.33.

The code transformation of *loop tiling* is given by Figure 2.33c. With this information, the following update rules are described:

$$\begin{aligned} L_X \langle X_{\min}, X_{\max} \rangle \rightarrow & L_X \left\langle \lfloor \frac{X_{\min}}{N} \rfloor, \lfloor \frac{X_{\max}}{N} \rfloor \right\rangle \\ & L_Y \langle 1, N \rangle \\ f_A \rightarrow & f_{A'} \end{aligned} \tag{2.25}$$

The transformation rule set does not lose flow information.

2.5.3 Vectorization optimizations

Vectorization is a compiler optimization that transforms a scalar implementation of a computation into a vector implementation [Nai04, RWY13]. It consists in processing multiple data at once, instead of processing a single data at a time. All silicon vendors now provide instruction set extensions for this purpose, and usually referred to as *single instruction, multiple data* (SIMD). Examples of SIMD instruction sets include Intel's MMX and iwMMXt, SSE (Streaming SIMD Extensions), SSE2, SSE3, SSSE3,

SSE4.x [INT] and AVX (Advanced Vector Extensions) [AVX], ARM's NEON technology [NEO], MIPS' MDMX (aka MaDMaX, MIPS Digital Media eXtension) [Gwe96] and MIPS-3D. The *vectorization factor* (VF) defines the number of operations that are processed in parallel, and is related to the size of the vector registers supported by the target architecture and the type of data elements. For 128-bit vectors (as in SSE and NEON), and the common types defined by the C language, VF ranges from 2 to 16.

Except our transformation framework, to our best knowledge there is no work studying the traceability of flow information and the impact of vectorization optimizations on WCET. So we introduce vectorization optimizations in detail.

Vectorization is a complex optimization that reorders computations. Certain conditions must be met to guarantee the legality of the transformation. Parallel loops, where all iterations are independent, can be obviously vectorized. More generally, a loop is vectorizable when the dependence analysis proves that the dependence distance is larger than the vectorization factor.

`for(i=0; i<n; i++)`
`{`
`  A[i]=B[i]+2;`
`}`
(a) loop-independent
	`for(i=1; i<n; i++)`
	`{`
	`  A[i]=A[i-1]+2;`
	`}`
	(b) loop-carried

Figure 2.34 – No dependence & loop-carried dependence

As an example, consider the loops of Figure 2.34.

- The loop on the left part of the figure is parallel (assuming arrays A and B do not overlap). The data elements written by an iteration are not written or read by any other iteration. This loop is safe to vectorize.
- Conversely, the loop shown on the right part of Figure 2.34 has a dependence of distance 1: values written at iteration i are read at iteration i+1. Processing these iterations in parallel would violate the dependence, and hence the loop cannot be vectorized.

Compilers usually contain a dependence test to identify the independent operations. For example, *LoopVectorizationLegality* in LLVM checks for the legality of the vectorization.

Modern vectorization technology includes two methods: Loop-Level Vectorization and *Superword Level Parallelism* (SLP) [LA00]. They aim at different optimization situations.

2.5.3.1 Loop-level vectorization

Loop-level vectorization operates on loops. In the presence of patterns where the same scalar instruction is repeatedly applied to different data elements in a loop, the loop-level vectorizer rewrites the loop with a single vector instruction applied to multiple

data. The number that determines the degree of parallelism of data elements is the vectorization factor. Figure 2.35 is an example of loop vectorization. VF in this example is 4, which means that in the vectorized version, four elements will be processed in one instruction in parallel[8]. In the meantime, the number of iterations is also divided by 4. Besides, an epilogue loop is created when the loop bound is not known at compile time to be a multiple of VF, to handle remaining iterations.

```
for(i=0; i<n; i++)
{
  A[i]=A[i]+2;
}
```

(a) Original source code

```
for(i=0; i<n-3; i+=4)
{
  A[i:i+3]=A[i:i+3]+2;
}
for(; i<n; i++)
{
  A[i]=A[i]+2;
}
```

(b) Vectorized

Figure 2.35 – Loop-level vectorization example

2.5.3.2 Superword level parallelism

```
typedef struct { char r, g, b } pixel;
void foo(pixel blend, pixel fg, pixel bg,
         float a)
{
  blend.r = a * fg.r + (1-a) * bg.r;
  blend.g = a * fg.g + (1-a) * bg.g;
  blend.b = a * fg.b + (1-a) * bg.b;
}
```

Figure 2.36 – The original code of superword level vectorization example

Superword level parallelism (SLP) focuses on basic blocks rather than loop nests. It combines similar independent scalar instructions into vector instructions. Consider the example of Figure 2.36. The three instructions in the function perform similar operations, only the operation elements are different (*r, g, b*). The SLP vectorizer analyzes these three instructions and their data dependencies, and combines them into a vector operation if possible.

2.5.3.3 Rule set

Rule set of loop-level vectorization The example of loop-level vectorization expressed at the source code level has been given in Figure 2.35. The corresponding

[8]Vectorization is performed at the intermediate code level. The example is given at source code level for readability. In the example $A[i:i+3]$ expresses that the four array elements at indices i to $i+3$ are processed in parallel.

modifications of the CFG are illustrated in Figure 2.37. In this figure, X in the left part is the loop bound of original L_x. The vectorization factor can be given as an input or decided by the compiler. Through vectorization, the scalar instructions within the loop body A are replaced with the corresponding vector instructions which constitute the new loop body A'.

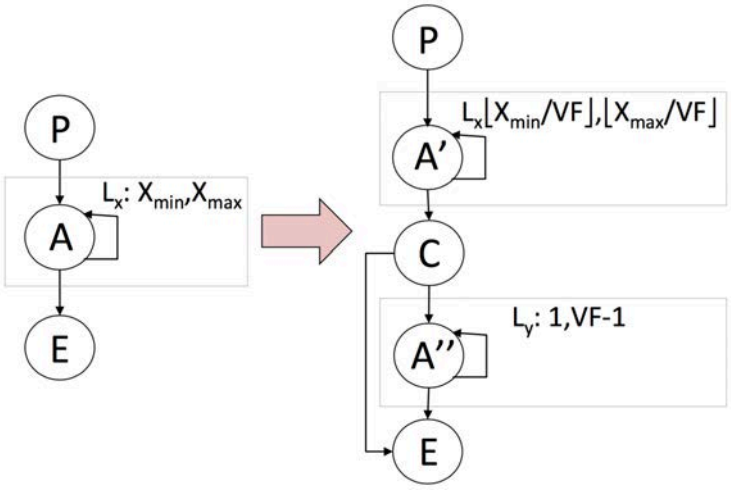


Figure 2.37 – The CFG of loop-level vectorization. The left part of the figure shows the original CFG, whereas the right part shows the vectorized one.

The example in this figure is a general case. When the loop bound is not known to be a multiple of VF, an epilogue loop is created to process the remaining iterations. Otherwise, the new loop is not needed, or equivalently the loop bound of the new loop is zero. Through the figure, we can observe that the structure of CFG (structure of loops, basic blocks) changes significantly, and the loop bounds of these two loops in the new CFG are also totally different from the original one.

Given the categorization of the transformation rules in Section 2.3, this optimization requires the application of both the change rule and the addition rule. The change rule is needed because of the change of the structure and loop bound of loop X. The addition of the new loop requires the addition rule. The set of rules describing the flow transformation of vectorization is given below.

$$L_X \langle X_{\min}, X_{\max} \rangle \rightarrow L_X \left\langle \lfloor \frac{X_{\min}}{VF} \rfloor, \lfloor \frac{X_{\max}}{VF} \rfloor \right\rangle$$
$$f_A \rightarrow VF \times f_{A'} + f_{A''}$$
$$L_Y \langle 1, VF - 1 \rangle$$

The first line is the change rule that expresses the change of the loop bounds of loop X. The loop bound of the original loop divided by the vectorization factor becomes the new one. The following line is change rule. The execution count f_A of node A should be replaced by $VF \times f_{A'} + f_{A''}$. The operations in A' are vector operations. The final

line is an addition rule that expresses the addition of loop bound for the newly created epilogue loop.

For the rule set of loop-level vectorization optimizations, no flow information is lost.

Rule set of superword level parallelism SLP transforms scalar instructions into vector instructions. Because SLP only focuses on basic blocks, it has no effect on the CFG. The transformation rule set for SLP is:

$$f_{BB\_scalar} \rightarrow f_{BB\_vector} \tag{2.26}$$

2.6 Related Work

WCET estimation methods have been the subject of significant research in the last two decades. Comparatively less research was devoted to WCET estimation with flow information traced from source-to-binary in the presence of optimizing compilers. In the rest of this section, we summarize the existing related research.

2.6.1 WCET estimation without or with "weak" optimizations

For the following WCET estimation methods, some of them do not support compiler optimizations, some of them do not support complex optimizations, and some of them support only one or a few kinds of optimizations.

An early approach is proposed by Park et al. [PS90, Par93, Cha94] (mentioned in Section 1.3). This approach uses an information description language (IDL) to perform the mapping from the object code to the source code, and performs timing analysis on the source code level. However, compiler optimizations are not supported appropriately by this approach. Our mechanism can handle code optimizations performed by a compiler.

Then a novel approach is presented by Engblom et al. [EEA98] to derive WCET when code optimizations are applied. A tool called co-transformer is designed in their work. It can transform timing information in parallel to the transformation of source code in the compilers. However, according to the authors, their data structures were not powerful enough to support some complex loop optimizations such as loop unrolling (They cannot represent the outcome which depend on the loop bounds). In contrast, as shown later, our mechanism can handle most LLVM [LA04, LLVb] optimizations, including loop unrolling.

Ferdinand et al. describe their WCET estimation method in [FHL+01]. Their method includes cache, pipeline, and path analysis. The method first reconstructs control flow with binary program and their analysis is based on this control flow. The annotations are needed for both the binary program and WCET analysis. So compared with our mechanism, the traceability of flow information and support of compiler optimizations in their method have to be done by the users.

The work of Lokuciejewski et al. [LFS+07] focuses on the effect of procedure cloning on WCET, because procedure cloning can improve the prediction of the code. Their

evaluation emphasizes the effect of procedure cloning on WCET. Our work also can enable procedure cloning, even with other optimizations. Then based on this work, Roeser et al. [RLF14] propose an approach optimizing for WCET, code size and energy consumption. The approach focuses on the ILP program and the trade-off of the multiple optimization criteria. On the contrary, we only focus on the WCET estimation and traceability of flow information.

Knoop et al. [KKZ11, KKZ12a] present a method to compute tight upper bounds of loop bounds. They refine program flows and rewrite different special loops into single-path *for* loops. They implement their method in the TuBound tool [PKST09]. However, their method just infers loop bounds for further WCET analysis, and only simple loop rewriting techniques are allowed.

Schoeberl et al. [SPPH10] propose a solution for a WCET analyzable Java system. Based on this WCET analysis tool, Hepp et al. [HS12] estimate WCET along with one of the effective compiler optimizations: inlining. Inlining is especially important for Java, where small setter and getter methods are considered good programming style. Our mechanism focuses on C/C++ instead of Java, and supports more compiler optimizations.

2.6.2 WCET estimation with compiler optimizations

The methods introduced in this subsection are most related to our work.

TuBound [oCEtIoCL, PSK08] is a WCET analysis tool combining the source code level annotations and low level WCET estimation with compiler optimizations. It enables the programmer to provide annotations at source code level. When the compiler optimizations are applied, it uses the transformation framework called FLOWTRANS to perform a source-to-source transformation. According to the different optimizations applied, this framework can transform the flow information. Based on TuBound, a new tool called r-TuBound [KKZ12b] is developed by Vienna University of Technology. r-TuBound can derive symbolic loop bounds automatically. Some loop bounds can only be analyzed by r-Tubound, not also by Tubound. TuBound and our mechanism both focus on the traceability of annotions. The difference is that our attention is on source-to-binary transformation instead of source-to-source transformation.

Kirner et al. [KP01, Kir02] describe a technique to provide path information inside the program for WCET analysis. They extend the C source code with additional syntax. With the additional syntax, scopes, loop limits and path information can be defined and these information can be recorded by the compilers. Then compilers match the information with the code of program. The match is traced by the compilers through transformations and optimizations.

Raymond et al. [RMPVC13] focus on timing analysis enhancement through traceability of flow information for synchronous programs. A method is introduced by them to improve timing analysis of programs generated from high-level synchronous design. Full traceability is guaranteed within the Lustre [CRT08] to C compiler, whereas error-prone graph matching is used so far for C to binary compilation. Our work is intended to complement theirs, with the overall objective of having *full* traceability of flow infor-

mation from very high level languages to binary code.

The SATIrE [SAT] system was introduced [Sch08] as a source-to-source analysis that integrates LLNL-ROSE source-to-source compiler infrastructure [Lab] and can generate analysis results and map them as source code annotations to the intermediate representation. By using ROSE's source-to-source transformation capabilities, it can add the analysis result of the nodes and edges as comments to the original source code. Afterwards, Barany et al. [BP10] use this system to build a WCET analysis tool which combines source-level analysis, optimization and a back-end compiler performing WCET analysis. The connection to several other timing analysis tools is also implemented in project ALL-TIMES aiming at improving and integrating existing European timing analysis tools [Bar]. Comparing with their source-to-source analysis, our method works on source-to-binary transformation.

Huber et al. [HPP13] propose an approach to relate intermediate code and machine code when generating machine code in compiler back-ends. The approach is based on a novel representation, called *control flow relation graph*, that is constructed from partial mapping provided by the compiler. In contrast to them, we focus in this thesis on optimizations performed at the intermediate code level.

The most related work is from Kirner et al. [KP03, Kir03, KPP10], who present a method to maintain correct flow information from source code level to machine code level. It transforms both manual annotations and platform-independent flow information in parallel to the code transformations performed by the compiler. We differ in the following respects. Their first implementation goes back to GCC-2.7.2, a compiler released in 1995, lacking a modern higher-level intermediate representation (GIMPLE was introduced much later [Mer03, GIM]), and featuring only "a small number of code transformations that change the control flow of a program significantly". We rely on state-of-the-art technology, and we can handle most optimizations in modern compilers. In a more recent implementation, Kirner et al. rely on source-to-source transformations, while we focus on traceability within the compiler, down to the code generator.

2.6.3 WCET estimation without traceability

The following mechanism can estimate WCET with compiler optimizations, but they do not trace flow information during the compilation.

Rodrigues et al. propose a mechanism called *back-annotation* in [RFdS11]. It establishes a link between the source code and the binary code with annotations and annotates the timing information on each source code line. At the end, the WCET analysis is at source code level. The same mechanism is also implemented in a WCC variant platform [PLM08]. But in this implementation, it links the high-level and intermediate representation level.

2.6.4 Optimizations for WCET

The following two methods focus on the different point of view from the above work and our work. They pay attention the WCET-aware optimizations. They detect, develop

and apply optimizations for WCET minimization.

Zhao et al. [ZKW+04, ZKW+] introduce an approach to reduce WCET by optimizing the code. The optimization starts from finding frequent paths to the worst-case paths in an application. Then by adapting and applying optimizations designed for these paths, the timing analyzer integrated within the compiler can calculate the timing result.

WCC [DAfESG] is a WCET-aware compiler that is different from modern compilers including a vast variety of optimizations mostly aiming at minimizing ACET. This specific compiler integrates optimizations for WCET minimization. These WCET-aware optimizations are applied only to minimize WCET aggressively, for example, Loop Nest Splitting [FS06], Loop Unrolling [LM09], Function Inlining [LGMM09] and so on. WCC is extended in [FK09] and [LF14]. The former one proposes a technique of SPM allocation of program code in WCET-aware optimization compilers. The latter one implements their approach in WCC and can determine the optimizations of tasks to achieve a schedulable system in hard real-time multitasking systems. Our work takes a different angle, by addressing general-purpose optimizations and compilers. Our experimental results show that most optimizations designed for average-case performance are also beneficial in the worst-case.

2.6.5 Vectorization research

Vectorization in compilers is necessary. The performance of vectorization has been studied extensively. However, to our best knowledge, there is no work studying the impact of vectorization on WCET. So in this subsection, we list the works on vectorization.

Maleki et al. [MGG+11] evaluate three main vectorizing compilers: GCC [GCC], the Intel C compiler [ICC] and the IBM XLC compiler [XLC]. They evaluate how well these three compilers vectorize benchmarks and applications. The experimental results indicate the functionality and limitation of these three compilers. And they also try to give a reason to the limitation.

The introduction and evaluation of vectorization optimization in LLVM is presented in [AUTb]. That document introduces the two vectorizers: the Loop Vectorizer and the SLP Vectorizer. Then it explains the usage of vectorizers in LLVM through examples, and gives performance numbers of the LLVM vectorizer on benchmark gcc-loops (introduced in Subsection 4.2.1) as compared to GCC.

Further evaluation of vectorization is given by Finkel in [Fin]. In this document, the benchmark TSVC (introduced in Subsection 4.2.1) is used to test the vectorization with LLVM as compared to the GCC vectorizer.

Compared with the above documents, we focus on the WCET estimate as a performance metric instead of average-case performance. Except our work, to our best knowledge there is no work studying the impact of vectorization on WCET estimate.

2.7 Summary

This chapter has presented the theoretic foundations which are needed by our transformation framework. Then we have described the details of our framework: the linear constraints, the transformation rules, the manipulations of the rule set, the influence on estimated WCET. Our transformation framework succeeds in tracing and transforming flow information from C level to binary level with compiler optimizations. Afterwards, the typical compiler optimizations and their corresponding rule sets were presented. At the end, we have introduced the scientific work related to our transformation framework and the scope of our thesis. However, for now, our work is just at theoretical level. We need to verify our transformation framework by implementing it and using experiments to evaluate it. So in the next two chapters, firstly, we will implement our transformation framework in a modern compiler; then we estimate WCET with the implementation and validate our framework by analyzing the results.

Chapter 3

Implementation of Traceability in the LLVM Compiler Infrastructure

In this chapter, we integrate the transformation framework in the LLVM compiler infrastructure [LA04, LLVb]. At the beginning of this chapter, we give an overview of the LLVM compiler infrastructure and other tools used in our implementation. Then, we explain our integration within the LLVM compiler infrastructure in detail. At the end, some specific features of optimizations in the LLVM are described.

Theoretically, our transformation framework transforms flow information expressed as linear constraints. However, considering the engineering work, for the implementation within the LLVM compiler, we focus on tracing only the local loop bounds which are the most important information for most programs and WCET estimation. The reasons of choosing loop bounds are:

- Loop bounds are the mandatory information for WCET estimation.
- Loop optimizations may affect loop bounds, and have an important influence on performances.

3.1 Required Tools

In this section, we introduce all tools used in our implementation.

3.1.1 WCET analysis tools

A short summary of WCET analysis tools is in Section 1.1.2. Detailed WCET analysis tool descriptions have been given by Wilhelm et al. [WEE+08]. Furthermore, a comparison of WCET tools with the same benchmarks and execution platforms is provided in [Gus06, HGB+08, vHHL+11]. Here, we only detail the two tools used in our implementation:

- Heptane: for performance evaluation.

- OTAWA: for the traceability of semantics information from high-level language to binary level in W-SEPT project.

3.1.1.1 Heptane

HEPTANE [CP00, IRI] static WCET estimation tool is an open-source static WCET analysis tool. The purpose of this tool is to obtain upper bounds of programs' execution times by analyzing code statically. It implements IPET (Implicit Path Enumeration Technique) and includes cache analysis techniques for many cache architectures. Its current implementation is for the C language with some restrictions. Its supported instruction sets include MIPS and Arm. Heptane can analyze the source and/or the binary code.

Heptane requires the user to give the maximum number of loop iterations through annotations in the source program or an external annotation file. Besides, with the annotations in source code, compiler optimizations are inhibited by default. Our modified LLVM toolchain can produce the optimized binary code which contains the information needed by Heptane to calculate WCET results (the binary production is in Section 3.3.5). The final WCET bound is computed using an external evaluation tool (lp_solve [LPS] or CPLEX [CPL]).

3.1.1.2 OTAWA

OTAWA [BCRS10, TRA] is an open-source tool from IRIT, W-SEPT partner, it is dedicated to binary code analysis and to WCET computation. The motivation of OTAWA is provide an open framework that could be used by researchers to implement their analyses and to combine them to already implemented ones. So we modify the LLVM compiler to output the binary containing the information needed by OTAWA for the WCET estimation.

OTAWA supports a large range of target architectures (*e.g.* PowerPC, ARM, Sparc or M68HCS). It is organized in independent layers that provide an abstraction of the target hardware and associated Instruction Set Architecture (ISA) as well as a representation of the binary (.elf) code under analysis. The modified LLVM outputs the final flow information into the binary code. The flow information is retrieved and analyzed by OTAWA, and is combined to determine the final WCET. The integer linear program defined by the IPET method is generated through the information and the target hardware. Finally, OTAWA invoke the lp_solve to solve this linear program and get the final WCET results.

3.1.2 Flow information extraction and formulation

We have mentioned different methods to extract flow information in Section 1.3. In our implementation, flow information is extracted automatically by using the tool named oRange which is from IRIT, W-SEPT partner. The extracted information is stored in the FFX format files.

3.1.2.1 oRange

oRange [BdMS08, dMBBC10] is a static analysis tool that is used to determine flow information including loop bounds. It works on the C source code and uses flow analysis and abstract interpretation techniques to derive contextual loop bounds (a loop in a function can have different bounds with each call context).

We have used oRange to extract loop bounds and send them to the modified LLVM. The modified LLVM conveys this original information to the binary code when the optimizations are applied.

3.1.2.2 FFX

FFX [BCdM$^+$12] - Flow Facts in XML is a portable WCET annotation language and an intermediate format for WCET analysis. It is an XML-based file format that is used to represent all kinds of flow information. The FFX format allows combining flow information from different high-level tools and decreases the implementation effort to integrate different WCET analysis tool chains. For example, FFX is used by OTAWA/oRange as their native annotation language, meanwhile, it is easy to combine FFX within our modified LLVM.

FFX offers means to store flow information on both source code and binary code levels. oRange produces FFX files as output. The modified LLVM uses it as input and outputs a binary code version after optimizations. This output is generic enough to be usable for a large range of WCET estimations tools, *e.g.* Heptane static WCET estimation tool and OTAWA WCET analysis toolchain.

3.2 The LLVM Compiler Infrastructure

We integrated the transformation rules described in Section 2.3 in the LLVM compiler infrastructure [LA04], version 3.4.

LLVM is a collection of modular and reusable compiler and toolchain technologies. The name of LLVM was an acronym for Low Level Virtual Machine. Nowadays, LLVM has developed and included a variety of subprojects, and it can handle different tasks. LLVM is written in C++. LLVM supports C and C++ originally, nevertheless theoretically it can handle programs written in arbitrary programming languages with a wide variety of front ends.

3.2.1 LLVM components

As shown in Figure 3.1, LLVM consists in a three-phase compiler.

clang is the first phase. It is a compiler front end for the C, C++, Objective-C and Objective-C++ programming languages. Clang parses, validates and diagnoses errors in the C/C++ code, and then translates the code into the LLVM Intermediate Representation (IR).

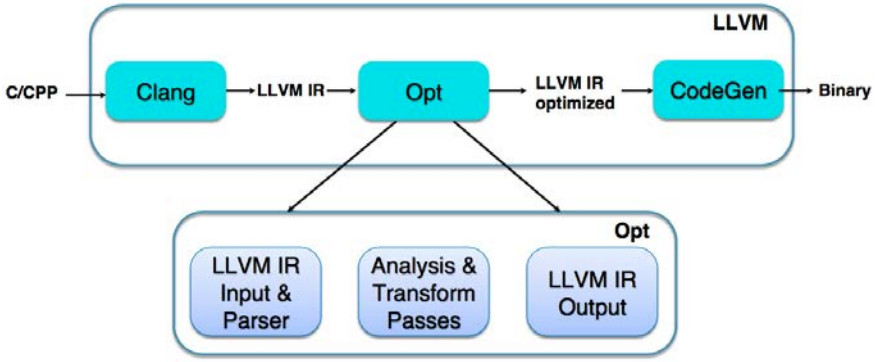


Figure 3.1 – LLVM Compiler Infrastructure

opt is the modular LLVM optimizer and analyzer. It takes LLVM intermediate representation as input and parses LLVM IR. With the objective of improving the code quality, it runs a series of specified analysis and optimizations on the IR.

codegen is the last phase - the compiler backend. It is a framework that provides a suite of reusable components for translating the LLVM intermediate representation to the machine code for a specified target. The output can be assembly form or binary machine code format. At version 3.4 LLVM supports many instruction sets, including ARM, MIPS, x86/x86-64, XCore and so on.

The LLVM IR [LLVa] is the core of LLVM. It is a low-level programming language and similar to assembly. It locates between high-level program and the low-level backend. It is the only interface to the optimizer and different components in LLVM. For LLVM IR, there are three isomorphic forms: an in-memory compiler IR, an on-disk bitcode representation, and a human readable assembly format.

3.2.2 Passes

LLVM is built around the notion of pass[1]. A pass performs an action on the program either to collect information or to transform the program. They consist in *Analysis* passes, *Transform* passes and *Utility* passes.

Analysis passes compute various information for program visualization purposes or that subsequent passes can use, such as dominator trees, alias analysis, or loop forests.

For example, *dot-cfg* is a pass to print the control flow graph into a .dot graph [DOT]; *scalar-evolution* is a pass to analyze and categorize scalar expressions in loops. The information obtained by this pass is useful for a transform pass *strength reduction*.

[1] http://llvm.org/docs/Passes.html

Transform passes use or invalidate the analysis passes. They all mutate the program more or less. Each pass can specify its impact on already available information. It may specify that particular information is preserved (allowing further passes to reuse it without recomputing it), while others are invalided, hence destroyed, and must be recomputed.

For example, *loop-reduce* is a transform pass which uses the information provided by *scalar-evolution* to perform a strength reduction on array references inside loops.

Utility passes are the passes whose utility do not fit categorization, *i.e.* are neither analysis nor transform passes.

For example, the pass *verify* is an utility pass. It is needed to verify an LLVM IR code after an optimization which is undergoing testing.

3.2.3 Supported optimizations

We implemented our transformation framework into the optimizations in LLVM compiler infrastructure. Figure 3.1 lists the supported optimizations in LLVM. They are almost all the transform LLVM optimizations of level (`-O3`). In Figure 3.2, we list the optimizations which are supported by our transformation framework, but not implemented in LLVM.

3.3 Implementation within the LLVM Compiler Infrastructure

In this section, we present all the details in the implementation within the LLVM compiler infrastructure. Figure 3.2 shows the structure of the whole implementation. Through observing the figure, you can find two major differences from the original LLVM. One is the addition of several external tools for the input and usage of flow information. Another is the modification of *opt*. We modified *opt* in four places: the addition of a new pass to store the annotations; another new pass to read in annotations from an external file; the individual transformations to convey annotations all the way down; and the IR writer to dump the updated annotations to a file. In the following of this section, we explain the details of these modifications.

3.3.1 External components

On Figure 3.2, there are two external components, depicted as yellow boxes: *loop bound estimation tool* and *WCET estimation tool*. The former derives loop bounds from the source code. Loop bounds are traced throughout the optimization passes. The *WCET estimation tool* calculates the WCET from the binary code, in which modified loop bounds have been generated. The corresponding tools in our implementation are oRange and Heptane/OTAWA. The input/output format of our framework is FFX. They were all described in the previous section.

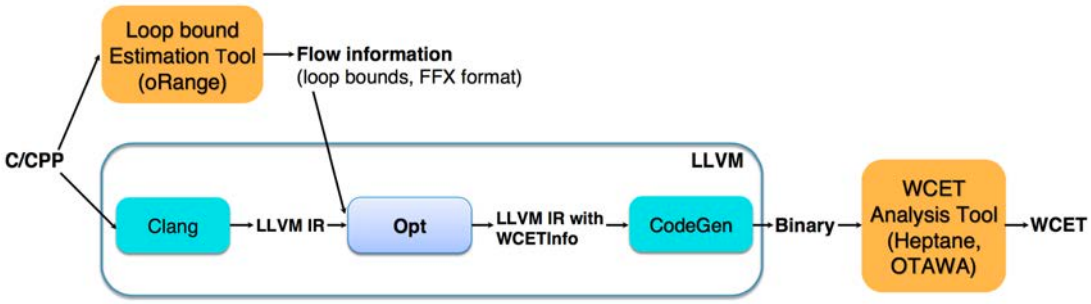


Figure 3.2 – Implementation of traceability in LLVM

3.3.2 Representation of flow information (WCETInfo)

We added to LLVM a new type of information, named *WCETInfo*, to be attached to the program. Its current purpose is to map loops (Loop objects in LLVM) to the corresponding estimate of loop bounds (both lower and upper loop bounds).

WCETInfo uses the "ImmutablePass" class for its implementation. Any new pass in LLVM should choose a superclass to subclass. We choose "ImmutablePass" class for our new pass because ImmutablePass is usually used for passes that only run once and are never invalidated. This is not a normal type of LLVM pass, but can provide information for other passes. WCETInfo needs run only once at the beginning, and avoid the loop bounds information being destroyed by the other passes.

3.3.3 Input of flow information

In order to read loop bounds generated by the oRange, we add another new pass into *opt*. Loop bounds extracted by oRange are expressed in FFX. The loop bounds in FFX are stored in order of their position in the function based on the source code. In LLVM, *LoopInfo* is a class designed to compute and store the loop information including natural loop identification, loop header, nesting structure of loops, loop depth and so on.

So our new pass, first reads the external annotation from FFX, then gets *LoopInfo*. By comparing the loop sequence in FFX and loop information in *LoopInfo*, we can obtain the corresponding loop bounds for each loop in LLVM and store these matching in WCETInfo.

As a side product of using automatically generated loop bounds, we were able to compare the loop bounds generated by oRange with those available inside LLVM using the *scalar-evolution* analysis pass. *scalar-evolution* implements the representation of scalar expressions, and can compute the execution count of a loop based on the expressions. All loop bounds calculated by *scalar-evolution* were also computed by oRange and the bounds were identical.

3.3.4 Transfer of flow information

Similarly to other information in LLVM, transformation passes may have one of the following behaviors with respect to WCETInfo:

Preserve WCETInfo. This is when the transformation does not modify loops, or when loops are modified, but we know that their bounds remain unchanged. Constant propagation is an example of this case.

Update WCETInfo. This happens when loops are modified, but we are able to apply the corresponding transformation to the loop bound information, according to one of the rules of Section 2.3. The example of this kind of transformation passes is loop interchange.

Add WCETInfo. This is needed when a new loop is added to the CFG by a transformation pass. If the loop bound of this new loop can be derived according the semantic of the optimization, we add the matching of the loop bound and new loop into WCETInfo. For example, if the called function contains loops, inline will introduce the loops into the caller function. Another example is loop distribution which divides the original loop into two.

Delete WCETInfo. This occurs when the transformation is unknown, or is known to be too complex to propagate loop bounds correctly. These optimizations are WCET unfriendly, and may render the WCET impossible to compute. Thus, they should be disabled from a compiler targeting real-time systems. An error will be generated if these optimizations are used.

The default for every pass is to delete the WCETInfo, as this is the safe behavior. Still use loop rerolling as the example, if we do not the tranformation of this optimization, and we preserve the WCETInfo, the WCET result is unsafe.

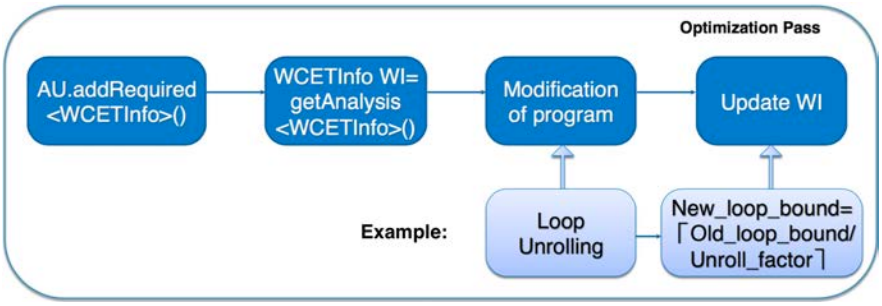


Figure 3.3 – Process of WCETInfo Update

The process of updating WCETInfo in transformation passes is depicted in Figure 3.3. Firstly, we add *AU.addRequired<WCETInfo>()* to each transformation pass which has an effect on WCETInfo. *AU.addRequired<>* means that this pass requires a previous pass to be executed. So by adding that code, the compiler knows that

this transformation pass needs WCETInfo and arranges for it to be run before this pass. Then *WCETInfo WI= getAnalysis <WCETInfo>()* is required. This method call returns a reference to WCETInfo. Then according to the transformation rules of each optimization introduced in Section 2.5, we operate the different behaviors on WCETInfo.

The example in Figure 3.3 is loop unrolling. The loop bound is modified in this optimization. So when a loop is unrolled, its loop bound in WCETInfo should be updated with the new one: $\lceil \frac{Old\_loop\_bound}{Unrolling\_Factor} \rceil$ (refer to Section 3.3.7.1).

The addition and modification of WCETInfo are operated manually. Because for each optimization, we need analyze the code to detect its impact on loop bounds and to find the appropriate location to add the modification code.

We verify the new loop bounds by comparing them with those available using the *scalar-evolution* analysis pass. All new loop bounds calculated by *scalar-evolution* are equal to the loop bounds in WCETInfo which are transformed by our transformation framework. So we can conclude that both our transformation framework and our implementation are correct.

3.3.5 Output of flow information

Code generation was also modified after all optimization passes to output the final loop bounds in the binary code in a specific section of the binary, for subsequent use in the WCET calculation.

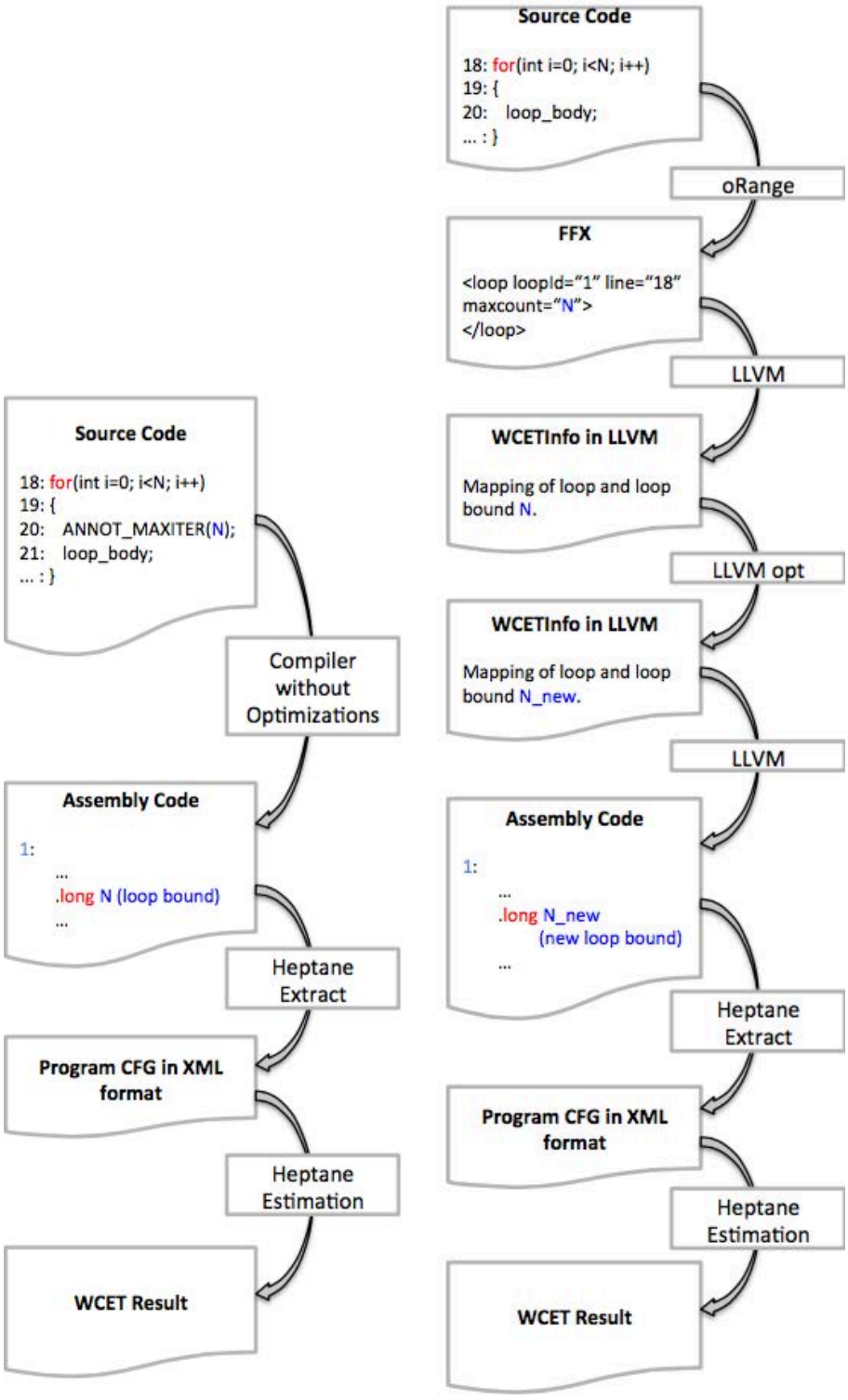
For Heptane, it can use the information in this binary directly. For OTAWA, we provide a tool to extract the final flow information and analyze the binary code. And the tool can combine all these information to constitute a FFX file which can provide enough information to OTAWA for WCET estimation.

3.3.6 The comparison with original Heptane estimation process

Figure 3.4 shows two different processes of WCET estimation by using Heptane.

Figure 3.4a presents the original Heptane working process. At first, manual annotations giving maximum number of iterations for loops is added in source code ($ANNOT\_MAXITER(N)$). After compilation without optimizations, these annotations are transferred into assembly code and then, the final binary files. The flow information extraction tool of Heptane called HeptaneExtract extracts the control flow graph including instruction addresses and loop bounds in the annotations from the binary files to an XML file. Then with this XML file, Heptane can analyze address and cache, calculate WCET by using an ILP formulation, even print the program in text or graphical format.

However, in our implementation (shown in Figure 3.4b), manual annotations are inessential. oRange can analyze and extract the loop bounds and store them into FFX. Actually, Heptane can use an external XML-based annotation file which plays a similar role to FFX. Both of them can be used interchangeable. Considering that FFX is generated by oRange automatically and is a part of W-SEPT project, we finally choose



(a) Original WCET estimation with Heptane

(b) WCET estimation in our implementation

Figure 3.4 – The WCET estimation process.

FFX as the annotation format. Then LLVM reads, retains and traces the loop bounds and also output them as the format which Heptane can analyze. The updated loop bounds are inside the optimized binary files. So Heptane can extract and analyze the binary files, and calculate the final WCET with the updated loop bounds.

3.3.7 Specific features of optimizations in LLVM

Some optimizations in LLVM have some specific features differing from the general version presented in Section 2.5. When we implement the transformation rules on these optimizations, we must rewrite them combining these specific features in LLVM.

Some examples about these difference are presented in the following:

3.3.7.1 Loop unrolling

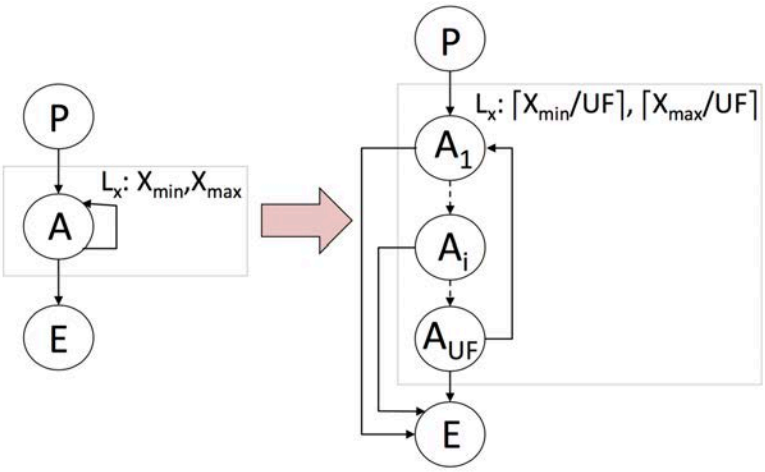


Figure 3.5 – The CFG of loop unrolling in LLVM. The left part of the figure shows the original CFG, whereas the right part shows the unrolled one.

Loop unrolling in LLVM is a little different from the one presented in Section 2.5.2.1. Figure 3.5 indicates the difference. The method in LLVM uses a conditional test inside the unrolled loop instead of the epilogue loop. This also results that the loop bound of the unrolled loop differs from the one in Figure 2.23.

With the new loop bound and conditional test in the new loop bodies, the transformation rules corresponding to Figure 3.5 are:

$$L_X \langle X_{\min}, X_{\max} \rangle \rightarrow L_X \left\langle \lceil \frac{X_{\min}}{UF} \rceil, \lceil \frac{X_{\max}}{UF} \rceil \right\rangle$$
$$f_A \rightarrow f_{A_1} + \ldots + f_{A_{UF}}$$

The first line is also a change rule that expresses the change of the loop bound of loop X. But, here the ceil of, instead of the floor of the loop bound of the original loop divided

by unrolling factor becomes the new one. Another difference is the remove of epilogue loop. Node A are replicated by unrolling. f_A should be replaced as $f_{A_1} + \ldots + f_{A_{UF}}$.

3.3.7.2 Vectorization optimization

Vectorization in LLVM has some specific features that slightly differ from those presented in Section 2.5.3. Actually, LLVM combines vectorization with loop unrolling introduced in Section 2.5.2.1 in a single pass. When transforming flow information, we must thus consider the effect of both transformations.

Unrolling applying jointly with vectorization can also generate more independent instructions. Vectorization as defined in LLVM is depicted in Figure 3.6.

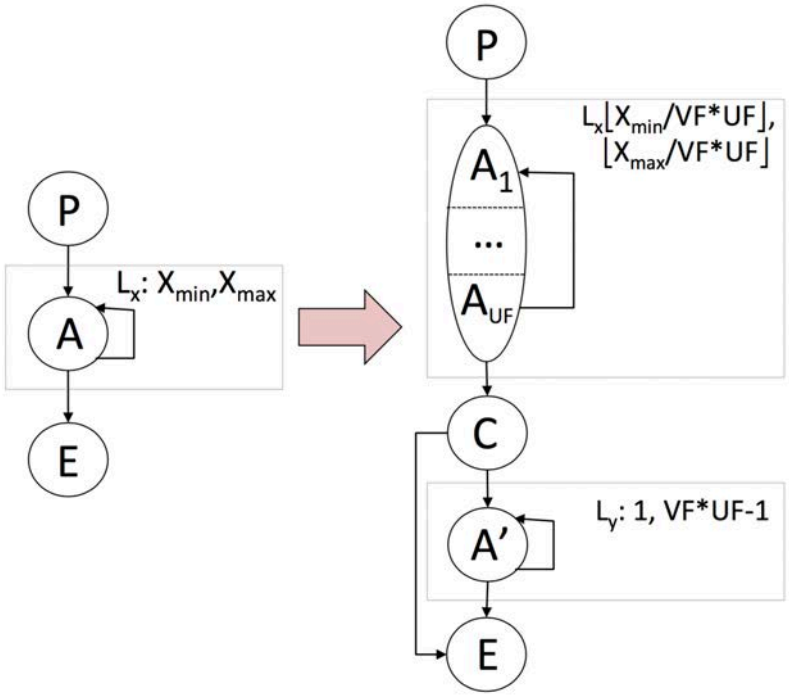


Figure 3.6 – The CFG of loop-level vectorization in LLVM. The left part of the figure shows the original CFG, whereas the right part shows the vectorized one.

Compared to Figure 2.37, the differences are that UF appears, because of the joint application of unrolling and vectorization. The loop body A is replicated $VF \times UF$ times for each loop iteration after the vectorization, and the new loop body $A_1 \ldots A_{UF}$ is the vector operations transformed from the original scalar loop body according to the vectorization factor. The transformation rules corresponding to Figure 3.6 are:

$$L_X \langle X_{\min}, X_{\max} \rangle \rightarrow L_X \left\langle \lfloor \frac{X_{\min}}{VF \times UF} \rfloor, \lfloor \frac{X_{\max}}{VF \times UF} \rfloor \right\rangle$$
$$f_A \rightarrow VF \times UF \times f_{A_1 \ldots A_{UF}} + f_{A'}$$
$$L_Y \langle 1, VF \times UF - 1 \rangle$$

The first line is also a change rule that expresses the change of the loop bound of loop X. But, here the loop bound of the original loop divided by the product of vectorization factor and unrolling factor becomes the new one. Another difference is about node $A_1 \dots A_{UF}$. This node contains vector operations which are replicated by unrolling. f_A should be replaced as $VF \times UF \times f_{A_1 \dots A_{UF}} + f_{A'}$.

3.4 Summary

The chapter presented the implementation of our transformation framework in LLVM compiler infrastructure. We introduced the different tools used within the implementation. Then the details about the storage and processing of flow information are given. And we also mentioned some specific features of optimizations in LLVM, *e.g.* loop unrolling and loop-level vectorization.

The final logical lines of code of our implementation within the LLVM compiler infrastructure is about 1500. I also developed a tool to extract flow information from the binary code generated by modified LLVM. Its lines of code is about 300.

The main goal of W-SEPT is to trace flow information through the compilation flow, from high-level language to C level and finally to binary level. Our transformation framework and implementation in LLVM compiler infrastructure accomplish the part from C level to binary level.

At this moment, we have succeeded in the implementation. So the next step should be the experiments. The experimental result is presented in the next chapter and we also give analyses on the result.

Optimization name	**Description**
Redundancy elimination, procedure, control-flow and low-level optimizations of LLVM	
adce	Aggressive dead code elimination
argpromotion	Promote "by reference" arguments to be "by value" arguments
constmerge	Merge duplicate global constants
correlated-propagation	Correlated value propagation
deadargelim	Deletes dead arguments from internal functions
dse	Intra basic-block elimination of redundant stores
early-cse	Early common subexpression elimination
functionattrs	Interprocedural deduction of function attributes
globalopt	Transforms simple global variables that never have their address taken
globaldce	Eliminate unreachable internal globals
inline	Replace a function call site with the body of the called function
instcombine	Combine instructions into less and simple instructions
ipsccp	Interprocedural conditional constant propagation
sccp	Sparse conditional constant propagation
jump-threading	Reduction of the number of branch instructions in case of chained branching
mem2reg	Promote memory reference to be register references
memcpyopt	Transformations related to eliminating calls to memcpy
prune-eh	Remove unused exception handling info
reassociate	Reassociate expressions to promote better constant propagation
simplifycfg	Dead code elimination and basic block merging
sroa	Scalar replacement of aggregates
strip-dead-prototypes	Strip unused function prototypes
tailcallelim	Elimination of tail recursion
Loop Optimizations of LLVM	
loop-simplify	Canonicalize natural loops to make subsequent analyses and transformations simpler and more effective
lcssa	Transform loops in closed SSA form
licm	Loop invariant code motion (move invariant code outside loop body)
loop-unswitch	Transforms loops that contain branches on loop-invariant conditions to have multiple loops
indvars	Canonicalize induction variables: analyzes and transforms the induction variables into simpler forms suitable for subsequent analysis and transformation
loop-idiom	Loop idiom recognizer: transforms simple loops into a non-loop form
loop-deletion	Deletion of loops with non-infinite computable trip counts that have no side effects and do not contribute to the computation of the function's return value
loop rotation	Replacement of a loop with the exit test at the start of a loop with an equivalent one, with the test at the end of the loop
loop-unroll	Replication of loop body by some unrolling factor to reduce branches and increase instruction-level parallelism
Vectorization Optimizations of LLVM	
loop-vectorize	Loop vectorizer: rewrite scalar instruction loop with a single vector instruction applied to multiple data
slp-vectorizer	Superword-level parallelism vectorizer: combine similar independent instructions into vector instructions

Table 3.1 – Supported optimizations included in LLVM.

Optimization name	**Description**
Other supported optimizations not implemented in LLVM	
if simplification	Removal of empty or not taken branches in conditional constructs
branch optimization	Elimination of branches
tail merging	Merge the instructions in different basic block targeting same destination into a new basic block
loop interchange	Exchange the order of two loops in a perfect loop nest. In general, it switches the outer loop to the inner position and vice versa
loop coalescing	Turn a nested loop into a single loop
loop collapsing	Less general version of loop coalescing
loop fission	Split a loop into multiple loops with the same iteration space as the original one and a subset of the original loop body
loop fusion	Rewrite multiple loops with the same loop bound into a single one
loop peeling	Move some iterations from the loop to outside of the loop
loop spreading	Move some computations from one loop to another
loop tiling	Divide the iteration space of a loop into tiles

Table 3.2 – Supported optimizations not implemented in LLVM.

Chapter 4

Experimental Evaluation of Traceability

In this chapter, we evaluate our transformation framework by experiments. Our purpose is to demonstrate that our transformation framework can trace flow information through compiler optimizations and allow us to calculate better WCET bounds. Through the implementation within the LLVM compiler infrastructure and the analysis results estimated by WCET analysis tools, we show the feasibility of our transformation framework.

This chapter consists of two parts:

- In Chapter 3, we mention that we have supported almost all the transform LLVM optimizations of level (`-O3`). However, because of lack of support of vectorization instructions in WCET analysis tools, we evaluate our transformation framework on the LLVM compiler optimizations of level (`-O3`) except for vectorization optimizations in Section 4.1.
- For the verification of vectorization, we relied on measurements on actual hardware. The measurements of vectorization optimizations and the analysis of measurement results are described in Section 4.2.

4.1 Experiments for Traceability without Vectorization

The objectives of the experiments are:

1. To verify the implementation of our transformation framework in LLVM compiler infrastructure;
2. To use the experimental results to find out the impact of compiler optimizations on estimated WCET.

4.1.1 Benchmarks

The evaluation of our transformation framework on WCET estimation requires a common set of benchmarks which serve a multitude of needs. The common choice is

Mälardalen benchmarks.

We demonstrate the impact of our mechanism on program optimization and annotation transformation with the standard set of WCET benchmarks from Mälardalen University [MBe].

Mälardalen benchmarks are collected from different research groups and tool vendors by Mälardalen WCET research group. They are used to evaluate and compare different WCET analysis tools and methods. The benchmarks are provided as C source files. Their descriptions, characteristics and lines of code are listed in Figure 4.1[1].

Benchmark	**Description**	**Comments**	**LOC**
bs	Binary search for the array of 15 integer elements.	Completely structured.	114
cnt	Counts non-negative numbers in a matrix.	Nested loops, well-structured code.	267
fdct	Fast Discrete Cosine Transform.	A lot of calculations based on integer array elements.	239
fibcall	Simple iterative Fibonacci calculation, used to calculate fib(30).	Parameter-dependent function, single-nested loop.	72
insertsort	Insertion sort on a reversed array of size 10.	Input-data dependent nested loop with worst-case of $(n^2)/2$ iterations (triangular loop).	92
jfdctint	Discrete-cosine transformation on a 8x8 pixel block.	Long calculation sequences (*i.e.*, long basic blocks), single-nested loops.	375
ludcmp	LU decomposition algorithm.	A lot of calculations based on floating point arrays with the size of 50 elements.	147
matmult	Matrix multiplication of two 20x20 matrices.	Multiple calls to the same function, nested function calls, triple-nested loops.	163
ndes	Complex embedded code.	A lot of bit manipulation, shifts, array and matrix calculations.	231
ns	Search in a multi-dimensional array.	Return from the middle of a loop nest, deep loop nesting (4 levels).	535
nsichneu	Simulate an extended Petri Net.	Automatically generated code containing large amounts of if-statements (more than 250).	4253
ud	Calculation of matrixes.	Loop nesting of 3 levels.	163

Table 4.1 – The descriptions, characteristics and lines of code of benchmarks used in our experiment

4.1.2 Target hardware

WCETs are estimated by using the Heptane timing analysis tool, implementing the Implicit Path Enumeration Technique (IPET) for WCET calculation. The ILP solver

[1]Information comes from http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

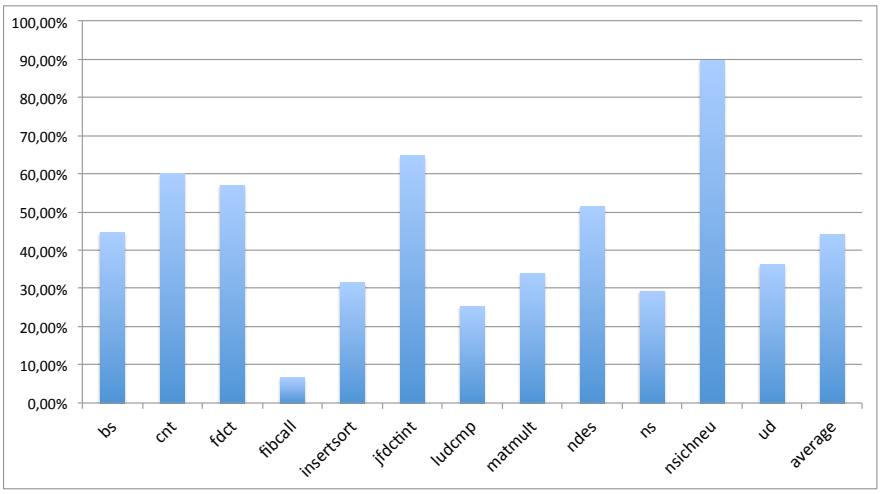


Figure 4.1 – Impact of optimizations (`-O3`) on WCET. The y-axis represents the WCET with optimizations, normalized with respect to the WCET without optimization (`-O0`)

is CPLEX [CPL]. For the scope of this document, to ease the understanding of results, a very simple hardware model is used by Heptane. A 32-MIPS processor is considered, with a 2-level hierarchy of caches and a perfect data cache. The L1 instruction cache is a 2-way 512-byte cache with 32-byte lines, and the L2 cache is a 8-way 16-Kbyte cache with 64-byte lines (the L1 cache size is voluntarily small to match the small size of Mälardalen benchmarks). The cache latency is set to 1 cycle for L1, 10 cycles for L2, and the memory latency to 50 cycles. Both cache levels implement LRU replacement. No instruction-level parallelism (pipeline) is assumed in the architecture.

Code is compiled to assembly using LLVM, version 3.4. The GNU assembler then compiles assembly code to binary that is used to feed Heptane. Optimized codes use the `-O3` option of LLVM (with the exception of vectorization optimizations which generate vectorization instructions not supported by WCET analysis tools yet. The experiment of vectorization optimizations in introduced in the Section 4.2.). Note that LLVM has most of its optimizations in `-O3`.

4.1.3 Impact of optimizations on estimated WCET

Figure 4.2 shows the impact of compiler optimizations on the estimated WCET computed by Heptane. Results are normalized with respect to the code compiled at `-O0`.

Firstly, the fact we are able to compute the WCET of all benchmarks (a single missing loop bound would make the computation impossible) shows that we are able to transform all flow information from C code to binary, and all loop bounds in the optimized binary are correct. The correctness is verified in both of the following two ways:

- Automatic way: we compare the loop bounds traced by our transformation framework with those available using the *scalar-evolution* analysis pass.

	bs	cnt	fdct	fibcall	insertsort	jfdctint	ludcmp	matmult	ndes	ns	nsichneu	ud	average
WCET (cycles) at -O3	359	5790	6130	62	976	7169	6255	105474	44805	6865	96617	6307	
deadargelim	-	-	-	-	-	-	-	-	-	-	-	-2 %	-0.2 %
early-cse	37 %	-	-1 %	-	33 %	-	12 %	-	-5 %	0.1 %	-	8 %	7 %
indvars	-	-10 %	-1 %	-	7 %	2 %	7 %	8 %	8 %	-	-	11 %	3 %
inline	20 %	-12 %	-	409 %	-	-	4 %	1 %	16 %	19 %	-	-	38 %
jump-threading	15 %	-	-	-	-	-	-	-	-	-	-	-	1 %
licm	-	-	-	-	-	-	-	-	-	-	-	3 %	0.3 %
loop-unroll	-	-32 %	-	-	-	-	-1 %	-	-5 %	-	-	-4 %	-3.5 %
mem2reg	-	-	-	-	-	-	-16 %	-	0.1 %	-	-	-2 %	-1.5 %
reassociate	-	-	-	-	-	-0.4 %	-	-	-	-	-	-	-0.03 %
simplifycfg	16 %	-	-	-	-	-	0.2 %	-	4 %	-	-	-1 %	1.6 %
sroa	-	-	-	-	-	-	-	-	16 %	-	-	-	1.3 %
adce	-	-	-	-	-	-	-	-	-	-	-	-	-
constmerge	-	-	-	-	-	-	-	-	-	-	-	-	-
correl.-prop.	-	-	-	-	-	-	-	-	-	-	-	-	-
dse	-	-	-	-	-	-	-	-	-	-	-	-	-
functionattrs	-	-	-	-	-	-	-	-	-	-	-	-	-
globaldce	-	-	-	-	-	-	-	-	-	-	-	-	-
globalopt	-	-	-	-	-	-	-	-	-	-	-	-	-
ipsccp	-	-	-	-	-	-	-	-	-	-	-	-	-
lcssa	-	-	-	-	-	-	-	-	-	-	-	-	-
loop-deletion	-	-	-	-	-	-	-	-	-	-	-	-	-
loop-idiom	-	-	-	-	-	-	-	-	-	-	-	-	-
loop-simplify	-	-	-	-	-	-	-	-	-	-	-	-	-
memcpyopt	-	-	-	-	-	-	-	-	-	-	-	-	-
prune-eh	-	-	-	-	-	-	-	-	-	-	-	-	-
sccp	-	-	-	-	-	-	-	-	-	-	-	-	-
tailcallelim	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 4.2 – Change in estimated WCET when one optimization is disabled (1-off). Reference is `-O3`. Positive numbers denote a beneficial effect of the optimization (WCET degrades when it is disabled).

- Manual way: we get the loop bounds from the LLVM IR (intermediate representation) and assembly code manually, and compare these with the ones traced by our framework.

Secondly, we observe that option `-O3` yields to an important reduction of estimated WCETs: 55 % in average, and up to 93 % (optimized WCET is 7 % of unoptimized) for benchmark *fibcall*, whose main function contains only a function call, can benefit from the inlining. The compiler can do more optimizations when inlining is applied.

4.1.3.1 Individual impact of optimizations (1-off)

We try to disable one of the optimizations to find out the impact of individual optimization on estimated WCET.

Disabling *instcombine* and *loop-rotate* causes problems to the compilation flow of LLVM. Further optimizations crash, and cause an abort of LLVM. Loop rotation transforms top-tested loops into bottom-tested loops. While this transformation has a marginal impact on performance, it is an enabler for others. We hypothesize that further optimizations assume this transformation has been applied, hence crash when we disable it. *Instcombine* also applies some normalization, such as moving constant operand of a binary operator to the right hand side. Again, further optimizations probably assume the IR is in a normal form and fail when it is not the case. Thus, we kept both optimizations enabled.

Table 4.2 reports our results. For each benchmark (horizontally), we report in the first row the WCET (in cycles) when all optimizations are enabled. Following rows report the estimated WCET when disabling each optimization individually. For example, in the bottom left corner, we see that disabling *simplifycfg* causes an increase of 16 % of WCET of *bs*. In other words, *simplifycfg* improves the estimate. On the contrary, it has an adversary impact on *ud* (next to last column): the WCET estimate is better by 2 % when *mem2reg* is not run. A dash sign means no change.

General Comments Disabling some optimizations has no impact on the estimated WCET. Some simply do not apply to our real-time benchmarks. For example: all loops compute useful values, hence loop deletion has nothing to do; *prune-eh* removes unused exception handlers, which do not exist in C code. Others, such as tail call elimination or memcopy optimization recognize specific patterns that do not occur in our benchmarks. *Globalopt* considers global variables whose addresses are never taken, and optimizes away constant and write-only variables.

Some optimizations, such as *loop-simplify*, do modify the code. It turns out that, in our configuration and for our benchmarks, the estimated WCET remains unaltered.

Register promotion is implemented by *mem2reg*. This is a key optimization that replaces costly memory accesses by much faster register uses. It is a priori surprising that turning it off does not result in major degradation. The reason is that *sroa* achieves the same effect. This is further discussed in Section 4.1.3.2.

Through the figure, we can observe that *inline* is an important optimization that can affect the estimated WCET of many benchmarks (more than half in the experiments). The reason for this is that *inline* replaces a function code into the body of called function. This can save the overhead of the function call, and make the further optimizations possible in the called function.

Common subexpression elimination (*early-cse*) and induction variable canonicalization (*indvars*) are basic optimizations of any compiler targeting Average Case Execution Time (ACET). Our results show that they also have a dramatic impact on estimated WCET. These two classic optimizations alone can improve the tightness of WCET by valuable amounts.

Code Layout and I-Cache Effects Some transformations result in a minor improvement or degradation (±2 % or so) of the estimated WCET. We suspected this could be a random effect due to a slightly different code layout, resulting in marginally different misses in the cache. To validate our hypothesis, we re-executed the entire experiments, allowing direct memory access (i.e. assuming a perfect cache). The new result is shown in Table 4.3. As expected we observed that these differences vanish, *e.g.*, *early-cse* on *ns*, *mem2reg* on *ud* and *deadargelim* on *ud*.

Scalar Replacement of Aggregates Disabling *sroa* only impacts *ndes*, but the effect is significant: it results in more than 16 % increase in estimated WCET. Visual

	bs	cnt	fdct	fibcall	insertsort	jfdctint	ludcmp	matmult	ndes	ns	nsichneu	ud	average
WCET (cycles) at -O3	109	2710	3010	2	656	4059	3865	104724	37420	6615	11327	3917	
deadargelim	-	-	-	-	-	-	-	-	-	-	-	-	-
early-cse	14 %	-	-1.5 %	-	41 %	-	10 %	-	0.3 %	-	-	8 %	6 %
indvars	-	0.6 %	-1 %	-	1 %	3 %	9 %	8 %	8 %	-	-	8 %	3 %
inline	2 %	20 %	-	8200 %	-	-	2 %	1 %	20 %	18 %	-	-	688 %
jump-threading	-5 %	-	-	-	-	-	-	-	-	-	-	-	-0.5 %
licm	-	-	-	-	-	-	-	-	-	-	-	5 %	0.4 %
loop-unroll	-	26 %	-	-	-	-	5 %	-	1 %	-	-	5 %	3 %
mem2reg	-	-	-	-	-	-	-27 %	-	-	-	-	-	-2.2 %
reassociate	-	-	-	-	-	-0.4 %	-	-	-	-	-	-	-0.03 %
simplifycfg	-	-	-	-	-	-	0.1 %	-	0.3 %	-	-	0.1 %	0.05 %
sroa	-	-	-	-	-	-	-	-	17 %	-	-	-	1.4 %
adce	-	-	-	-	-	-	-	-	-	-	-	-	-
constmerge	-	-	-	-	-	-	-	-	-	-	-	-	-
correl.-prop.	-	-	-	-	-	-	-	-	-	-	-	-	-
dse	-	-	-	-	-	-	-	-	-	-	-	-	-
functionattrs	-	-	-	-	-	-	-	-	-	-	-	-	-
globaldce	-	-	-	-	-	-	-	-	-	-	-	-	-
globalopt	-	-	-	-	-	-	-	-	-	-	-	-	-
ipsccp	-	-	-	-	-	-	-	-	-	-	-	-	-
lcssa	-	-	-	-	-	-	-	-	-	-	-	-	-
loop-deletion	-	-	-	-	-	-	-	-	-	-	-	-	-
loop-idiom	-	-	-	-	-	-	-	-	-	-	-	-	-
loop-simplify	-	-	-	-	-	-	-	-	-	-	-	-	-
memcpyopt	-	-	-	-	-	-	-	-	-	-	-	-	-
prune-eh	-	-	-	-	-	-	-	-	-	-	-	-	-
sccp	-	-	-	-	-	-	-	-	-	-	-	-	-
tailcallelim	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 4.3 – Change in estimated WCET when one optimization is disabled (1-off). Reference is `-O3`. Without cache effects.

inspection confirms that this benchmark makes intensive use of small structs (of two and three elements) that can easily be promoted.

As mentioned, *sroa* also captures the register promotion, but this effect is visible when both optimizations are turned off (see Section 4.1.3.2).

Loop Unrolling Loop unrolling degrades *cnt* (-32 %), *ludcmp* (-1 %), *ndes* (-5 %) and *ud* (-4 %). With perfect I-cache, loop unrolling is always worthwhile, improving *cnt, ludcmp, ndes* and *ud* respectively by 26 %, 5 %, 1 %, and 5 %.

Loop unrolling is well known to compiler developers to be a double-edged sword. Average performance improves as long as the working set stays in the cache. The additional misses in the instruction cache cancel the benefits of the optimization. Our results show that the same holds for estimated WCET. In the case of *cnt*, the reason is slightly different. The loop is unrolled ten times (fully unrolled). Its new size is about 400 bytes, which fits in the L1 I-cache. However, due to the structure of the application, there is no reuse of this code. The increased number of cycles comes from additional cold misses. Note, though, that in case of reuse, additional capacity misses are expected, because the unrolled loop size is close to the cache size (512 bytes), and most of its contents is evicted.

	bs	cnt	fdct	fibcall	insertsort	jfdctint	ludcmp	matmult	ndes	ns	nsichneu	ud
WCET (cycles) at -O3	359	5790	6130	62	976	7169	6255	105474	44805	6865	96617	6307
mem2reg + sroa	70 %	20 %	59 %	848 %	187 %	47 %	150 %	161 %	57 %	49 %	20 %	143 %
early-cse + indvars	37 %	10 %	-3 %	-	34 %	2 %	20 %	9 %	2 %	0.1 %	-	21 %
simp.cfg + early-cse	37 %	-12 %	-1 %	-	34 %	-	13 %	-	-5 %	0.1 %	-	11 %

Table 4.4 – Change in estimated WCET when two optimizations are disabled (2-off). Reference is `-O1`. Positive numbers denote a beneficial effect of the optimizations (WCET degrades when they are disabled).

4.1.3.2 Combined impact of optimizations (2-off)

As a final experiment without vectorizations, we disabled pairs of optimizations out of `-O3`. We want to find out the impact of combined optimizations on estimated WCET. We tried all pairs. Given the amount of data (more than 300 optimization combinations on 12 benchmarks), we only report highlights in Table 4.4.

Firstly, we focus on two optimizations *early-cse*, *indvars* and their combinations, we often observe additive effects between them, and this observation is generally true for our set of optimizations and benchmarks. For example, for *ludcmp*, the estimated WCET increases 20 % when *early-cse* (individual effect: 12 %) and *indvars* (individual effect: 7 %) are both disabled.

As for ACET, optimizations are not always additive. This is also true for WCET. It can be illustrated by the pair *simplifycfg + early-cse*. We can observe from the figure: *Bs* (37 % vs. 37 % & 16 %), w*cnt* (-12 % vs. - & -), *ud* (11 % vs. 8 % & -1 %).

As anticipated in the previous section, *sroa* and *mem2reg* have overlapping effects. As mentioned in the LLVM documentation, *sroa* also performs *alloca promotion*, which serves the purpose of SSA formation and results in an effect similar to register promotion. Disabling both optimizations results is a considerable degradation of the WCET estimate.

4.2 Experiments for Traceability with Vectorization

The previous section mentions that we estimate WCET at the `-O3` level of LLVM compiler except for vectorization optimizations. The reason is that the WCET estimation tools we have access to (Heptane and Otawa) do not currently support SIMD instruction sets. We thus relied on measurements on actual hardware to collect real execution times. Note that we use measurements only due to the lack of support for SIMD instructions. Loop bound information is correctly traced through the compiler in all cases. We have manually verified that the loop bounds traced by our framework are the same as the loop bounds after vectorization. Meanwhile, we have done the automatic verification by using the *scalar-evolution* analysis pass.

We want to use the measurements to show that vectorization does not only reduce average-case execution times but also reduces worst-case execution times. The experimental results in this section do not directly validate our transformation framework and implementation on vectorization optimizations, but rather motivate the need to

use vectorization in hard real-time systems.

4.2.1 Benchmarks for vectorization

Vectorization optimizations do not benefit much from Mälardalen benchmarks, we introduce some other special benchmarks for vectorization optimizations. For vectorization optimization, we evaluate its impact by using the two following benchmark suites:

- **TSVC.** TSVC stands for Test Suite for Vectorizing Compilers, developed by Callahan, Dongarra and Levine [CDL88]. It contains 135 loops. It has been extended by the Polaris Research Group at the University of Illinois at Urbana-Champaign [Pol] and contains 151 loops now. In this paper, we use the modified version included in the LLVM distribution [TSV]. We restricted our experiments to the 112 single-path programs of TSVC.

- **Gcc-loops.** Gcc distributes a set of loops collected on the GCC vectorizer example page [Auta]. It is now also one of the LLVM test suites and we can test it in LLVM compiler [Nuz]. We restricted our experiments to the 15 single-path programs in this benchmark suite.

These two benchmark suites are test suites for vectorizing compilers and as such are loop-intensive programs.

Our benchmarks were restricted to single-path programs to guarantee that we are not impacted by path coverage issues. Despite using only single-path benchmarks, we still cannot equal the result to WCET. Because, in some benchmarks, there is a type of memory addressing called indirect addressing. Indirect addressing means that when the benchmarks want to read value from memory or write result to memory, they need through an indirection. The form is like `a[b[i]]=...` (this pattern is called scatter) or `...=a[b[i]]` (this pattern is called gather). In this type of memory addressing, different data can impact the execution time. The benchmarks *s491*, *s4112*, *s4113*, *s4114*, *s4117*, *vag* and *vas* include indirect addressing. So we remove these benchmarks from our list of experimental single-path test cases. In this way, without the impact of input data, indirect addressing and different paths, the execution time of these rest single-path benchmarks can be considered as WCET.

4.2.2 Environment

We ran each benchmark five times, on an otherwise unloaded machine, and we report the highest observed value. Observed execution times are very stable: the standard deviation for most benchmarks is less than 0.3 s (for the benchmarks whose runtime ranges from 8 s to 50 s) and 1 s (for the benchmarks whose runtime ranges from 50 s to 650 s) on ARM. It is less than 0.03 s (for the benchmarks whose runtime ranges from 1.8 s to 10 s) and 0.1 s (for the benchmarks whose runtime ranges from 10 s to 475 s) on Intel.

We selected the following two different architectures for measuring WCETs:

ARM: For the first target, we choose a Panda Board[2] equipped with an OMAP4 ARMv7 processor (v7l) running at 1.2 GHz. It features the advanced SIMD ISA extension NEON. The NEON vectors used in our experiment are 128-bit vectors. The size of L1 instruction cache and data cache are both 32 KB. The size of L2 cache per core is 1 MB. The operating system is Ubuntu 12.04.5.

Intel: We also experimented with an Intel architecture: experiments were performed on an Intel Core i7-3615QM CPU with four cores running at 2.30 GHz. The CPU instruction set include extensions SSE4.1 and SSE4.2 (Streaming SIMD Extensions 4), and AVX. The version used in our experiment is SSE4.2, and the vector size is 128-bit. The size of L1 instruction cache and data cache are both 32 KB. The size of L2 cache per core is 256 KB and L3 cache is 6 MB. The operating system is Mac OS X 10.10.1. Besides, we made sure to turn off Turbo Boost to guarantee the same execution circumstances for every measurement [ERS+14] (Turbo Boost Technology can automatically allow processor cores to run faster than the rated operating frequency if they are operating below power, current, and temperature specification limits).

The Intel Architecture usually is not used in real-time systems. We use it only to denote the effect of vectorization optimizations on different architectures and do not claim it is predictable enough to be used in real-time systems.

Execution times in our experiments are measured using the C library function *clock()*, that returns the number of clock ticks elapsed since the program was launched.

In LLVM, VF (vectorization factor) and UF (unrolling factor) can be specified by the user or decided by the compiler. The latter is better in most situations, because they are selected by using a cost model. So in the following experiments, we let LLVM choose VF and UF.

4.2.3 Impact of vectorization on WCET

We first measure the WCET obtained with all LLVM optimizations at level `-O3` (which enables the vectorizer) for TSVC on ARM and Intel architectures. Then, we evaluate the impact of vectorization optimization by manually disabling the vectorization (`-O3 -fno-vectorize`) (in this situation loop unrolling is still enabled).

Figures 4.2 and 4.3 report on WCET improvements for respectively TSVC on ARM and TSVC on Intel. Reported numbers are the WCET improvement ratios brought by vectorization ($WCET_{no-vec}/WCET_{vec}$). There is one bar per TSVC benchmark; the X-axis gives the number of the benchmark. Here, not all single-path benchmarks are shown. Only the benchmarks which are affected by vectorization on ARM or Intel architecture are presented. Each figure also reports the average WCET improvement ratio for all benchmarks (including those not affected by vectorization).

It immediately appears that the WCET of many TSVC benchmarks does not improve when turning on vectorization (WCET improvement ratio is 1, hence not shown

[2] http://pandaboard.org

on figure). This comes from the nature of TSVC, whose objective is to stress vectorizing compilers. Many kernels could simply not be vectorized by LLVM, regardless of our addition for traceability. In those cases, the vectorizer fails, and flow information is simply left unmodified.

4.2.3.1 TSVC and ARM Architecture

Figure 4.2 shows the impact of vectorization on WCET for the ARM architecture and single-path benchmarks in TSVC.

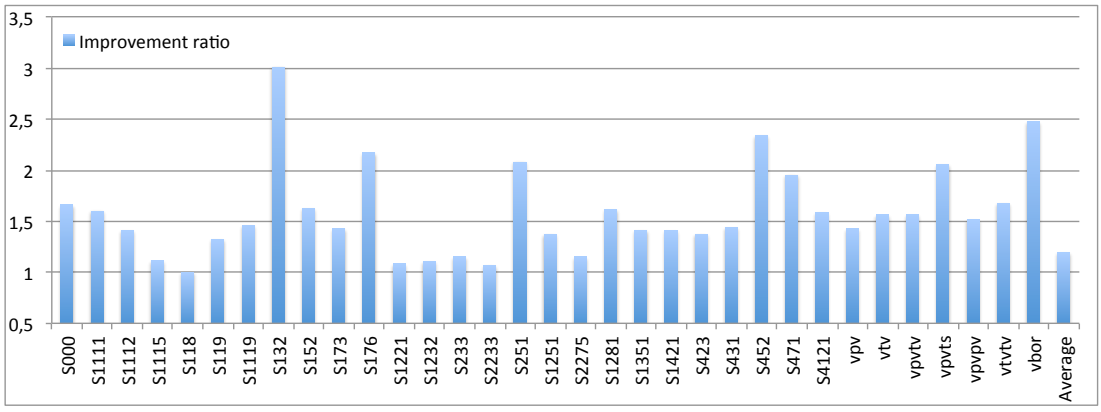


Figure 4.2 – Impact of vectorization on WCET (ARM, single-path TSVC benchmarks). The y-axis represents the WCET improvement ratio brought by vectorization: $WCET_{no-vec}/WCET_{vec}$.

Table 4.5 shows the description of the benchmarks in Figure 4.2.

As mentioned before, not all benchmarks benefit from vectorization. Only a little more than 1/4 of them have a significant WCET improvement ratio, averaging 1.19×.

Theoretically, the WCET improvement ratio could reach 4: these benchmarks manipulate arrays of type `float`, and the NEON instruction set can operate four elements at the same time. However, the results show that the improvement ratio is around 2 in most cases. The main factors limiting performance are related to the memory subsystem: cache misses and available bandwidth. Vectorized code needs to load four times more data for a similar computational intensity, sometimes reaching the maximum physical bandwidth. And when arrays are larger than a cache level (typically L2 in our benchmarks), frequent cache misses also dominate the performance, limiting the improvement ratio.

4.2.3.2 TSVC and Intel Architecture

We run the same experiments on the Intel architecture. As Figure 4.3 shows, the LLVM vectorizer results in higher WCET improvement ratio on Intel; the average WCET improvement ratio is 1.44× compared with 1.19× on ARM.

Overall, the WCET improvement ratio due to vectorization is larger on Intel than on ARM. For most benchmarks, the improvement ratio is closer to 4. However, Intel

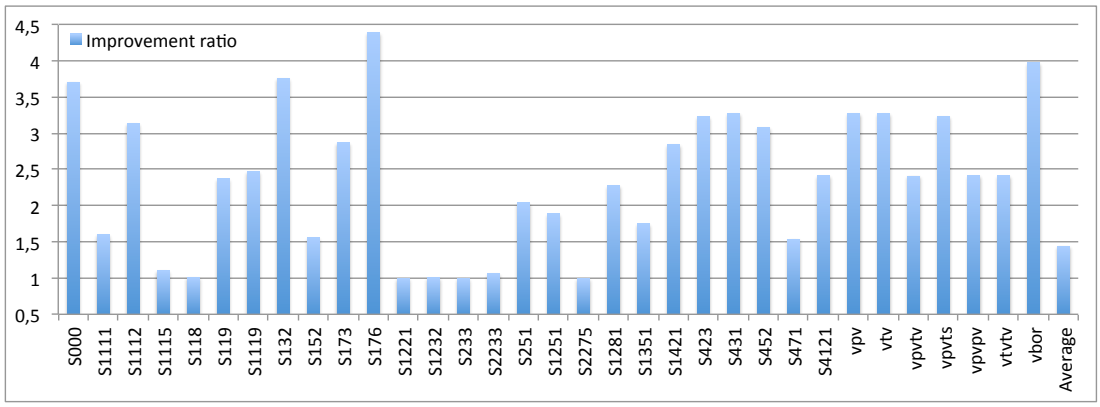


Figure 4.3 – Impact of vectorization on WCET (Intel, single-path TSVC benchmarks). The y-axis represents the WCET improvement ratio brought by vectorization: $WCET_{no-vec}/WCET_{vec}$.

is occasionally impacted by the same factors as ARM. In *s251*, *s1251* and other similar benchmarks, the performance increase is limited by the cache size: in these benchmarks, the overall size of all accessed arrays is larger than the L2 cache. Finally, the improvement ratio of *s176* is above 4: the loop is perfectly vectorized, and LLVM also applied loop unrolling, further increasing performance. We manually disabled loop unrolling and observed that the ratio drops below 4, as expected.

4.2.3.3 Gcc-loops

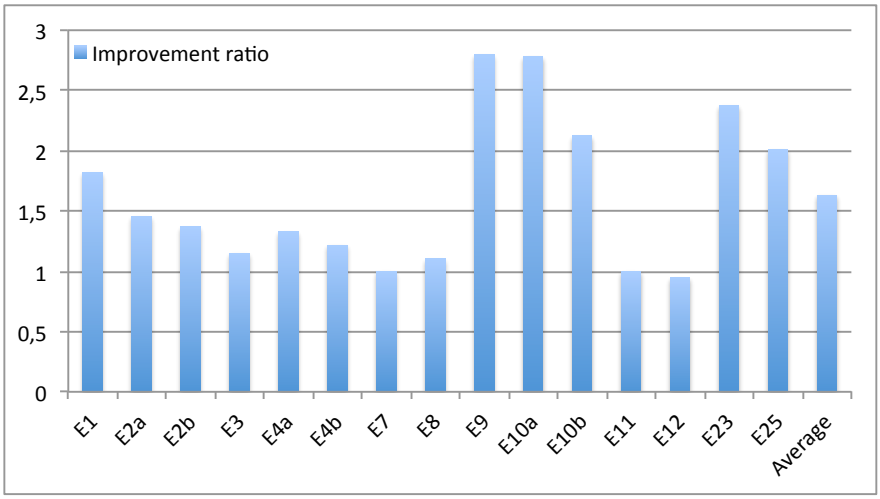


Figure 4.4 – Impact of vectorization on WCET (ARM, single-path Gcc-loops)

We measure the WCETs on the single-path codes from Gcc-loops on both Intel and ARM architecture. Results are presented on Figure 4.4 (ARM) and Figure 4.5 (Intel). As before there is one bar per benchmark in the benchmark suite, and the y-axis gives

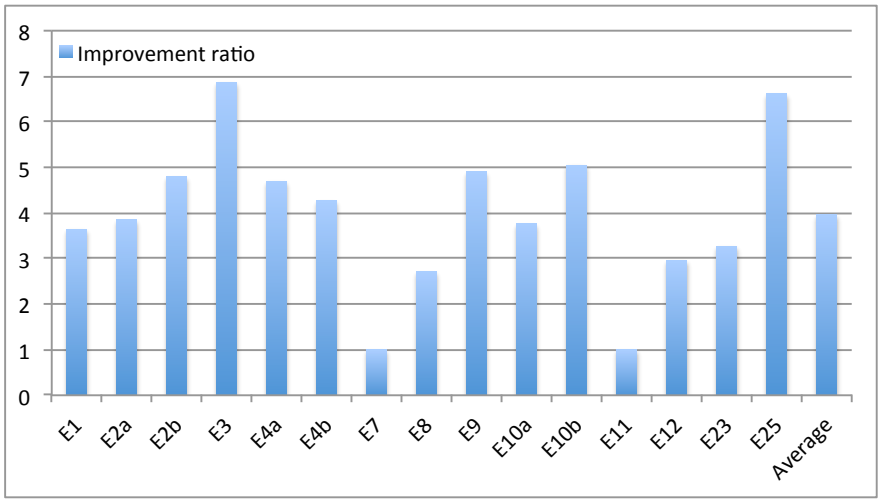


Figure 4.5 – Impact of vectorization on WCET (Intel, single-path Gcc-loops)

the WCET improvement ratio obtained when turning on vectorization.

In Figure 4.5, we can observe that there are 7 benchmarks whose improvement ratio is above 4. Except for *E25*, the reason for these benchmarks is the same as *s176* in TSVC: loop unrolling is applied and increases performance. In the case of *E25*, the high ratio is also due to a particularly poor sequential code which can be confirmed with the Intel Architecture Code Analyzer [Int13]: the generated sequential code results in many more micro-operations than the vectorized loop.

Through these figures, we can make similar observations as on TSVC. Vectorization reduces WCETs, and does this more effectively on Intel architecture.

4.3 Summary

This chapter presents the experimental results on different benchmarks and the analysis about them. The experimental results show that with our transformation framework, many optimizations even including inline and vectorization can be turned on. When the structure of CFG is modified by the optimizations, the annotation provided at source code level can be maintained and updated. Our transformation framework can make sure that no flow information is lost, even in the presence of CFG restructuring transformations such as loop unrolling and vectorizations. We also provide insight about the advantage of running particular optimization of the well accepted Mälardalen benchmarks. At the same time, the experimental results with measured WCETs show that vectorization is highly beneficial for real-time systems.

Loop name	**Description**
s000	linear dependence testing
s1111	jump in data access
s1112	linear dependence testing, loop reversal
s1115	linear dependence testing, triangular saxpy loop
s118	linear dependence testing, potential dot product recursion
s119	linear dependence testing
s1119	linear dependence testing
s132	global data flow analysis, loop with multiple dimension ambiguous subscripts
s152	interprocedural data flow analysis, collecting information from a subroutine
s173	symbolics, expression in loop bounds and subscripts
s176	symbolics, convolution
s1221	run-time symbolic resolution
s1232	loop interchange, interchanging of triangular loops
s233	loop interchange, interchanging with one of two inner loops
s2233	loop interchange, interchanging with one of two inner loops
s251	scalar and array expansion, scalar expansion
s1251	scalar and array expansion, scalar expansion
s2275	loop distribution is needed to be able to interchange
s1281	crossing thresholds, index set splitting, reverse data access
s1351	induction pointer recognition
s1421	storage classes and equivalencing, equivalence - no overlap
s423	storage classes and equivalencing, common and equivalenced variables - with anti-dependence
s431	parameters, parameter statement
s452	intrinsic functions, seq function
s471	call statements
s4121	statement functions, elementwise multiplication
vpv	control loops, vector plus vector
vtv	control loops, vector times vector
vpvtv	control loops, vector plus vector times vector
vpvts	control loops, vector plus vector times scalar
vpvpv	control loops, vector plus vector plus vector
vtvtv	control loops, vector times vector times vector
vbor	control loops, basic operations rates, isolate arithmetic from memory traffic
Loops containing indirect addressing	
s491	indirect addressing on lhs, store in sequence, scatter is required
s4112	indirect addressing, sparse saxpy, gather is required
s4113	indirect addressing, indirect addressing on rhs and lhs, gather and scatter is required
s4114	indirect addressing, mix indirect addressing with variable lower and upper bounds, gather is required
s4117	indirect addressing, seq function
vag	control loops, vector assignment, gather is required
vas	control loops, vector assignment, scatter is required

Table 4.5 – Loops in TSVC (only vectoried single-path)

Conclusion

Designers of hard real-time systems are required to compute the WCET of the components of their systems. This is accomplished by combining information provided at high level by programmers (e.g. loop bound information) and generated at low level by compilers. This combination is possible if a mapping is maintained between high- and low-level representations. Optimizing compilers typically break this simple mapping, and in order to keep this mapping and estimate WCET, developers usually turn all optimizations off.

The performance brought by modern compiler optimizations is demanded by more and more systems. Turning optimizations off is not a long-term policy. A solution is needed to estimate WCET with compiler optimizations.

We propose a transformation framework that traces flow information through compiler optimizations. The transformation framework consists of basic transformation rules which manipulate the flow information. Through these basic rules, we analyze most general optimizations and describe their corresponding rule sets.

A simplified version of our transformation framework was integrated within the LLVM compiler infrastructure. In this implementation, we focus on loop bounds. The implementation and the corresponding experimental results demonstrate that our transformation framework can trace and transform flow information when many compiler optimizations are turned on. We also provide insight about the advantage of running particular optimizations of the well accepted Mälardalen benchmarks. With our implementation, the estimated WCET can be derived in the presence of almost all general optimizations in LLVM.

Three years of PhD study have flown by. There are still a lot of ongoing work.

- For now, our transformation framework needs to analyze each optimization, and also needs to analyze the source code of the compiler optimizations to implement our framework. So maybe we need to define a more general and powerful transformation framework which generates transformation rule set and implement the rule set inside the code automatically.

- In Section 2.2.2, we mentioned that there is loss of information during the encoding of infeasible paths in a loop or in a function called multiple times. We have not found a better solution, and this may be a direction for future studies.

- In the experimental chapter, we mentioned that vectorized code cannot be ana-

lyzed by WCET analysis tools because of the lack of support for vector instruction set. A static WCET analysis of vectorization optimizations is needed to provide a further proof for our transformation framework.

- The implementation of our transformation framework is a simplified version. Only local loop bounds are traced within the optimizations. Global loop bounds can provide more precise loop bounds when for instance triangular loops are analyzed. This work needs two parts: support by WCET analysis tools and implementation in LLVM. Both of these two parts are not trivial, so this work is challengeable but significant.

- Considering the W-SEPT ANR project, we need to trace the flow information in the C code which is generated by compiler from high level languages such as Lustre. This kind of C code has a characteristic: there are many mutually exclusive branches. So extending traceability beyond loop bound information to mutually exclusive branches is one important future work.

- Introducing contextual information into the implementation is another important future work. For example, the loops in different calls may have different loop bounds. The achievement of contextual information can make the precision of loop bounds get a further improvement.

- Now, our implementation can output flow information for different WCET analysis tools (for now, at least for Heptane and OTAWA, and theoretically almost any WCET tools). However, when more flow information is included, it is not a trivial task. We should make the output of our implementation usable for different WCET analysis tools.

Bibliography

[Abs] AbsInt. aiT Worst-Case Execution Time Analyzers. http://www.absint.com/ait/.

[All70] Frances E Allen. Control flow analysis. In ACM Sigplan Notices, volume 5, pages 1–19. ACM, 1970.

[APT00] Hassan A Aljifri, Alexander Pons, and Moiez A Tapia. Tighten the computation of worst-case execution-time by detecting feasible paths. In Performance, Computing, and Communications Conference, 2000. IPCCC'00. Conference Proceeding of the IEEE International, pages 430–436. IEEE, 2000.

[Auta] Auto-vectorization in GCC. https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

[AUTb] Auto-vectorization in LLVM. http://llvm.org/docs/Vectorizers.html.

[AVX] Introduction to Intel Advanced Vector Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[Bar] Gergö Barany. SATIrE within ALL-TIMES: Improving timing technology with source code analysis. In Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2009), page 230.

[BCdM+12] Armelle Bonenfant, Hugues Cassé, Marianne de Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. FFX: a portable WCET annotation language. In Proceedings of the 20th Int'l Conference on Real-Time and Network Systems, RTNS '12, pages 91–100, New York, NY, USA, 2012. ACM.

[BCP02] Guillem Bernat, Anotione Colin, and Stefan M Petters. WCET analysis of probabilistic hard real-time systems. In Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE, pages 279–288. IEEE, 2002.

[BCP03] Guillem Bernat, Antoine Colin, and Stefan Petters. pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems. REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS, 2003.

[BCRS10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In Software Technologies for Embedded and Ubiquitous Systems, volume 6399 of Lecture Notes in Computer Science, pages 35–46. Springer Berlin Heidelberg, 2010.

[BdMS08] Armelle Bonenfant, Marianne de Michiel, and Pascal Sainrat. oRange: A tool for static loop bound analysis. In Workshop on Resource Analysis, University of Hertfordshire, Hatfield, UK, volume 9, page 08, 2008.

[BLH14] Bernard Blackham, Mark Liffiton, and Gernot Heiser. Trickle: automated infeasible path detection using all minimal unsatisfiable subsets. In Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th, pages 169–178. IEEE, 2014.

[BMB10] Adam Betts, Nicholas Merriam, and Guillem Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. In WCET, pages 54–63, 2010.

[BP10] G. Barany and A. Prantl. Source-level support for timing analysis. In Conference on Leveraging Applications of Formal Methods, Verification, and Validation, pages 434–448, 2010.

[BPSM+98] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML). World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210, 16, 1998.

[CDL88] David Callahan, Jack Dongarra, and David Levine. Vectorizing compilers: A test suite and results. In Proceedings of the 1988 ACM/IEEE conference on Supercomputing, pages 98–105. IEEE Computer Society Press, 1988.

[Cha94] Roderick Chapman. Worst-case timing analysis via finding longest paths in SPARK Ada basic-path graphs. University of York, Department of Computer Science, 1994.

[Che87] Moyer Chen. A Timing Analysis Language-(TAL). Department of Computer Science, University of Texas, Austin, TX, USA, 1987.

[CMPVR14] Hugues Cassé, Claire Maiza, Catherine Parent-Vigouroux, and Pascal Raymond. Schedulability and modular analysis: how to fit timing model? In OPRTC, 2014.

[CP00] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. Real-Time Systems, 18(2-3):249–274, 2000.

[CPL] IBM ILOG CPLEX Optimization Studio–High-performance mathematical programming engine. `http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud/`.

[CRT08] Paul Caspi, Pascal Raymond, and Stavros Tripakis. Synchronous programming. Handbook of Real-Time and Embedded Systems, 2008.

[DAfESG] Technical University of Dortmund Design Automation for Embedded Systems Group. WCET-AWARE COMPILATION. `http://ls12-www.cs.tu-dortmund.de/daes/en/research/wcet-aware-compilation.html`.

[dMBBC10] M. de Michiel, A. Bonenfant, C. Ballabriga, and H. Cassé. Partial Flow Analysis with oRange (short paper). In Symposium On Leveraging Applications of Formal Methods, Verification and Validation, number 6416 in LNCS, pages 479–482, 2010.

[DOT] The dot language. `http://www.graphviz.org/doc/info/lang.html`.

[EEA98] J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating worst-case execution times analysis for optimized code. In Euromicro Workshop on Real-Time Systems, pages 146–153, 1998.

[ERS+14] Laurel Emurian, Arun Raghavan, Lei Shao, Jeffrey M. Rosen, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. Pitfalls of accurately benchmarking thermally adaptive chips. In Workshop on Duplicating, Deconstructing, and Debunking (WDDD), June 2014.

[FH04] Christian Ferdinand and Reinhold Heckmann. aiT: Worst-case execution time prediction by static program analysis. In Building the Information Society, pages 377–383. Springer, 2004.

[FHF07] Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. Static memory and timing analysis of embedded systems code. In Proceedings of VVSS2007-3rd European Symposium on Verification and Validation of Software Systems, 23rd of March, pages 07–04, 2007.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In Embedded Software, pages 469–485. Springer, 2001.

[Fin] Hal Finkel. Autovectorization with LLVM. `http://llvm.org/devmtg/2012-04-12/Slides/Hal_Finkel.pdf`. 2012.

[FK09] Heiko Falk and Jan C Kleinsorge. Optimal static wcet-aware scratchpad allocation of program code. In Proceedings of the 46th Annual Design Automation Conference, pages 732–737. ACM, 2009.

[FLT06] Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. Design of a WCET-aware C compiler. In Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia, pages 121–126. IEEE Computer Society, 2006.

[FS06] Heiko Falk and Martin Schwarzer. Loop nest splitting for WCET-optimization and predictability improvement. In Embedded Systems for Real Time Multimedia, Proceedings of the 2006 IEEE/ACM/IFIP Workshop on, pages 115–120. IEEE, 2006.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. WCET, 15:136–146, 2010.

[GCC] GCC, the GNU Compiler Collection. `https://gcc.gnu.org`.

[GEL06] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Algorithms for infeasible path calculation. In WCET. Citeseer, 2006.

[GEL+09] J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg, and L. Källberg. ALF–a language for WCET flow analysis. In Proc. 9th Int'l Workshop on Worst-Case Execution Time Analysis (WCET2009)(Dublin, Ireland, pages 1–11, 2009.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International, pages 57–66. IEEE, 2006.

[GIM] Gimple online docs. `https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`.

[GN72] Robert S. Garfinkel and George L. Nemhauser. Integer programming, volume 4. Wiley New York, 1972.

[Gus06] Jan Gustafsson. The worst case execution time tool challenge 2006. In Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on, pages 233–240. IEEE, 2006.

[Gwe96] Linley Gwennap. Digital, mips add multimedia extensions. Microprocessor Report, 10(15):24–28, 1996.

[HGB+08] Niklas Holsti, Jan Gustafsson, Guillem Bernat, et al. WCET tool challenge 2008: report. 2008.

[HGKL14] Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Combining Bound-T and SWEET to Analyse Dynamic Control Flow in Machine-Code Programs. 2014.

[HPP13] B. Huber, D. Prokesch, and P. Puschner. Combined WCET Analysis of Bitcode and Machine Code Using Control-flow Relation Graphs. In Conference on Languages, Compilers and Tools for Embedded Systems (LCTES), pages 163–172, 2013.

[HS02] Niklas Holsti and Sami Saarinen. Status of the Bound-T WCET tool. Space Systems Finland Ltd, 2002.

[HS12] Stefan Hepp and Martin Schoeberl. Worst-case execution time based optimization of real-time java programs. In Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2012 IEEE 15th International Symposium on, pages 64–70. IEEE, 2012.

[HW02] C. A. Healy and D. B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. IEEE Trans. on Software Engineering, 28(8), 2002.

[ICC] Intel Compilers. `https://software.intel.com/en-us/intel-compilers`.

[INT] Using Intel Streaming SIMD Extensions and Intel Integrated Performance Primitives to Accelerate Algorithms. `https://software.intel.com/en-us/articles/using-intel-streaming-simd-extensions-and-intel-integrated-performance-primitives-to-accelerate-algorithms`.

[Int13] Intel Corporation. Intel Architecture Code Analyzer – User's Guide, 2.1 edition, 2013.

[IRI] IRISA. Heptane (Hades embedded processor timing analyzer) static WCET estimation tool. `https://team.inria.fr/alf/software/heptane`.

[Kir02] R. Kirner. The programming language wcetC. Technische Universität Wien, Institut für Technische Informatik, 2002.

[Kir03] R. Kirner. Extending optimising compilation to support worst-case execution time analysis. PhD thesis, Technische Universität Wien, 2003.

[KKP+07] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel. WCET analysis: The annotation language challenge. In PostWorkshop Proceedings of the 7th Int'l Workshop on WorstCase Execution Time Analysis, pages 83–99, 2007.

[KKP+08] Raimund Kirner, Albrecht Kadlec, Adrian Prantl, Markus Schordan, and Jens Knoop. Towards a Common WCET Annotation Language: Essential Ingredients. In Raimund Kirner, editor, 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. also published in print by Austrian Computer Society (OCG) under ISBN 978-3-85403-237-3.

[KKP+11] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. Software & Systems Modeling, 10(3):411–437, 2011.

[KKZ11] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. An Evaluation of WCET Analysis using Symbolic Loop Bounds. na, 2011.

[KKZ12a] J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. Perspectives of Systems Informatics, pages 227–242, 2012.

[KKZ12b] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. r-TuBound: Loop bounds for WCET analysis (tool paper). In Logic for Programming, Artificial Intelligence, and Reasoning, pages 435–444. Springer, 2012.

[Kou96] Apostolos A Kountouris. Safe and efficient elimination of infeasible execution paths in wcet estimation. In Real-Time Computing Systems and Applications, 1996. Proceedings., Third International Workshop on, pages 187–194. IEEE, 1996.

[KP01] Raimund Kirner and Peter Puschner. Transformation of path information for WCET analysis during compilation. In Real-Time Systems, 13th Euromicro Conference on, 2001., pages 29–36. IEEE, 2001.

[KP03] Raimund Kirner and Peter Puschner. Timing analysis of optimized code. In Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003). Proceedings of the Eighth International Workshop on, pages 100–105. IEEE, 2003.

[KPP10] R. Kirner, P. Puschner, and A. Prantl. Transforming flow information during code optimization for timing analysis. Real-Time Systems, 45(1):72–105, 2010.

[KPW04] Raimund Kirner, Peter Puschner, and Ingomar Wenzel. Measurement-based worst-case execution time analysis using automatic test-data generation. na, 2004.

[KS86] Eugene Kligerman and Alexander D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. Software Engineering, IEEE Transactions on, (9):941–949, 1986.

[LA00] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets, volume 35. ACM, 2000.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In Int'l Symp. on Code Generation and Optimization (CGO), pages 75–88, San Jose, CA, USA, March 2004.

[Lab] Lawrence Livermore National Laboratory. ROSE: an open source compiler infrastructure. `http://rosecompiler.org`.

[LCFM09] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In Int'l Symp. on Code Generation and Optimization (CGO), 2009.

[LF14] Arno Luppold and Heiko Falk. Schedulability-oriented wcet-optimization of hard real-time multitasking systems. Proceedings of JRWRTC, pages 9–12, 2014.

[LFS+07] Paul Lokuciejewski, Heiko Falk, Martin Schwarzer, Peter Marwedel, and Henrik Theiling. Influence of procedure cloning on WCET prediction. In Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis, pages 137–142. ACM, 2007.

[LGMM09] Paul Lokuciejewski, Fatih Gedikli, Peter Marwedel, and Katharina Morik. Automatic WCET reduction by machine learning based heuristics for function inlining. In 3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART), pages 1–15, 2009.

[Lis14] Björn Lisper. SWEET–A tool for WCET flow analysis. In Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, pages 482–485. Springer, 2014.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. Science of Computer Programming, 69(1):56–67, 2007.

[LLVa] LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`.

[LLVb] The LLVM Compiler Infrastructure. `http://llvm.org`.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In ACM SIGPLAN Notices, volume 30, pages 88–98. ACM, 1995.

[LM09] Paul Lokuciejewski and Peter Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for WCET minimization. In Real-Time Systems, 2009. ECRTS'09. 21st Euromicro Conference on, pages 35–44. IEEE, 2009.

[LPR14] Hanbing Li, Isabelle Puaut, and Erven Rohou. Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation. In Proceedings of the 22nd International Conference on Real-Time Networks and Systems, page 97. ACM, 2014.

[LPR15] Hanbing Li, Isabelle Puaut, and Erven Rohou. Tracing Flow Information for Tighter WCET Estimation: Application to Vectorization. In 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications RTCSA. IEEE, 2015.

[LPS] lpsolve: a Mixed Integer Linear Programming solver. `http://lpsolve.sourceforge.net`.

[MACT89] Aloysius Mok, Prasanna Amerasinghe, Moyer Chen, and Kamtron Tantisirivat. Evaluating tight execution time bounds of programs by annotations. IEEE Real-Time Systems Newsletter, 5(2-3):81–86, 1989.

[MBe] Mälardalen WCET Benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[Mer03] Jason Merrill. Generic and Gimple: A new tree representation for entire functions. In Proceedings of the 2003 GCC Developers' Summit, pages 171–179, 2003.

[MGG+11] Saeed Maleki, Yaoqing Gao, María Jesús Garzaran, Tommy Wong, and David A Padua. An evaluation of vectorizing compilers. In Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on, pages 372–382. IEEE, 2011.

[Nai04] Dorit Naishlos. Autovectorization in GCC. In Proceedings of the 2004 GCC Developers Summit, pages 105–118, 2004.

[NEO] The ARM NEON general-purpose SIMD engine. `http://www.arm.com/products/processors/technologies/neon.php`.

[Nuz] Dorit Nuzman. GCC-loops benchmarks for LLVM. `https://llvm.org/svn/llvm-project/test-suite/trunk/SingleSource/UnitTests/Vectorizer/gcc-loops.cpp`.

[oCEtIoCL] Institute of Computer Engineering and the Vienna University of Technology the Institute of Computer Languages. TuBound in Compiler Support for Timing Analysis Project. `http://costa.tuwien.ac.at/tubound.html`.

[oCNE] Institute of Computer and Braunschweig University of Technology Network Engineering. SymTA/P Tool of TU Braunschweig. `https://www.ida.ing.tu-bs.de/research/projects/symtap/`.

[oS] National University of Singapore. Chronos reserch prototype. `http://www.comp.nus.edu.sg/~rpembed/chronos/`.

[Par93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. Real-Time Systems, 5(1):31–62, 1993.

[PKST09] Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level WCET analysis. arXiv preprint arXiv:0903.2251, 2009.

[PLM08] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. A retargetable framework for multi-objective WCET-aware high-level compiler optimizations. In Proceedings of The 29th IEEE Real-Time Systems Symposium (RTSS) WiP, pages 49–52, 2008.

[PM14] Saeed Parsa and S. Mehdi. A XML-Based Representation of Timing Information for WCET Analysis. Journal of mathematics and computer science, 8:205–214, 2014.

[Pol] Polaris Research Group, University of Illinois at Urbana-Champaign. Extended test suite for vectorizing compilers. `http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz`.

[PPH+13] Peter Puschner, Daniel Prokesch, Ben Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on, pages 1–8. IEEE, 2013.

[PS90] Chang Park and Alan C Shaw. Experiments with a program timing tool based on source-level timing schema. In Real-Time Systems Symposium, 1990. Proceedings., 11th, pages 72–81. IEEE, 1990.

[PS97] Peter P Puschner and Anton V Schedl. Computing maximum task execution times–a graph-based approach. Real-Time Systems, 13(1):67–91, 1997.

[PSK08] Adrian Prantl, Markus Schordan, and Jens Knoop. Tubound–a conceptually new tool for worst-case execution time analysis. CHRISTIAN-ALBRECHTS-UNIVERSITAT KIEL, page 117, 2008.

[Rapa] RapiTime Explained White Paper. `http://www.rapitasystems.com/system/files/RapiTime%20Explained.pdf`.

[Rapb] Rapita Systems Ltd. Measurement-based WCET analysis tool RapiTime. http://www.rapitasystems.com/products/rapitime.

[Ray14] Pascal Raymond. A general approach for expressing infeasibility in implicit path enumeration technique. In Embedded Software (EMSOFT), 2014 International Conference on, pages 1–9. IEEE, 2014.

[RFdS11] Vıtor Rodrigues, Mário Florido, and Simao Melo de Sousa. Back annotation in action: from wcet analysis to source code verification. Actas of CoRTA, 2011.

[RLF14] Nicolas Roeser, Arno Luppold, and Heiko Falk. Multi-criteria optimization of hard real-time systems. Proceedings of the JRWRTC, 2014.

[RMPVC13] P. Raymond, C. Maiza, C. Parent-Vigouroux, and F. Carrier. Timing analysis enhancement for synchronous program. In Int'l Conference on Real-Time and Network Systems (RTNS), pages 141–150, 2013.

[RtiV] Sweden Research team in Vasteras. SWEET (SWEdish Execution Time tool) . http://www.mrtc.mdh.se/projects/wcet/sweet/.

[RWY13] Erven Rohou, Kevin Williams, and David Yuste. Vectorization technology to improve interpreter performance. ACM Trans. Archit. Code Optim., 9(4):26:1–26:22, January 2013.

[SAT] SATIrE: Static Analysis Tool Integration Engine. http://www.complang.tuwien.ac.at/satire/.

[Sch08] M. Schordan. Source-To-Source Analysis with SATIrE - an example revisited. In Scalable Program Analysis, number 08161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[SPPH10] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. Software: Practice and Experience, 40(6):507–542, 2010.

[tcr] Time-predictable Multi-Core Architecture for Embedded Systems. http://www.t-crest.org.

[Tid] Tidorum Ltd. Bound-T time and stack analyser. http://www.bound-t.com.

[TRA] TRACES, IRIT, Université Paul Sabatier. OTAWA: Open Tool for Adaptive WCET Analyses. http://www.otawa.fr.

[TSV] Test Suite for Vectorizing Compilers for LLVM. https://llvm.org/svn/llvm-project/test-suite/trunk/MultiSource/Benchmarks/TSVC/.

[vHHL+11] Reinhard von Hanxleden, Niklas Holsti, Björn Lisper, Erhard Ploedereder, Reinhard Wilhelm, Armelle Bonenfant, Hugues Cassé, Sven Bünte, Wolfgang Fellger, Sebastian Gepperth, et al. Wcet tool challenge 2011: Report. In Procs 11th Int Workshop on Worst-Case Execution Time (WCET) Analysis, 2011.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem–overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst., 7(3):36:1–36:53, 2008.

[XLC] IBM XL C and C++ Compilers family. `http://www-03.ibm.com/software/products/en/ccompfami`.

[ZKW+] Wankang Zhao, William Kreahling, David Whalley, Christopher Healy, and Frank Mueller. Improving WCET by optimizing worst-case paths. In Real Time and Embedded Technology and Applications Symposium, RTAS 2005. 11th IEEE, pages 138–147.

[ZKW+04] Wankang Zhao, Prasad Kulkarni, David Whalley, Christopher Healy, Frank Mueller, and Gang-Ryung Uh. Timing the WCET of embedded applications. In Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE, pages 472–481. IEEE, 2004.

List of Figures

List of Tables

Résumé

Les systèmes temps-réel sont omniprésents, et jouent un rôle important dans notre vie quotidienne. Pour les systèmes temps-réel dur, calculer des résultats corrects n'est pas la seule exigence, il doivent de surcroit être produits dans un intervalle de temps borné. Connaitre le pire cas de temps d'exécution (WCET - Worst Case Execution Time) est nécessaire, et garantit que le système répond à ses contraintes de temps. Pour obtenir des estimations de WCET précises, des informations de flot sont nécessaires. Ces informations sont généralement ajoutées via des annotations au niveau du code source, tandis que l'analyse de WCET est effectuée au niveau du code binaire. L'optimisation du compilateur est entre ces deux niveaux et a un effet sur la structure du code et sur les annotations.

Nous proposons dans cette thèse une infrastructure logicielle de transformation, qui pour chaque optimisation transforme les annotations du code source vers le code binaire. Cette infrastructure est capable de transformer les annotations sans perte d'information de flot.

Nous avons choisi LLVM comme compilateur pour mettre en œuvre notre infrastructure. Et nous avons utilisé les jeux de test Mälardalen, TSVC et gcc-loop pour démontrer l'impact de notre infrastructure sur les optimisations du compilateur et la transformation d'annotations. Les résultats expérimentaux montrent que de nombreuses optimisations peuvent être activées avec notre système. Le nouveau WCET estimé est meilleur (plus faible) que l'original. Nous montrons également que les optimisations du compilateur sont bénéfiques pour les systèmes temps-réel.

Abstract

Real-time systems have become ubiquitous, and play an important role in our everyday life. For hard real-time systems, computing correct results is not the only requirement. In addition, the worst-case execution times (WCET) are needed, and guarantee that they meet the required timing constraints. For tight WCET estimation, annotations are required. Annotations are usually added at source code level but WCET analysis is performed at binary code level. Compiler optimization is between these two levels and has an effect on the structure of the code and annotations.

We propose a transformation framework for each optimization to trace the annotation information from source code level to binary code level. The framework can transform the annotations without loss of flow information.

We choose LLVM as the compiler to implement our framework. And we use the Mälardalen, TSVC and gcc-loops benchmarks to demonstrate the impact of our framework on compiler optimizations and annotation transformation. The experimental results show that with our framework, many optimizations can be turned on, and we can still estimate WCET safely. The estimated WCET is better than the original one. We also show that compiler optimizations are beneficial for real-time systems.