Politecnico di Milano

V Facoltà di Ingegneria

Corso di laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Method specialization for Common-Intermediate-Language in a dynamic compiler

Relatore: Prof. Stefano CRESPI REGHIZZI
Correlatore: Ing. Simone CAMPANONI

Tesi di Laurea di:
Stefano ANELLI Matr. 711309

Anno Accademico 2008-2009

Ai miei genitori

Ringraziamenti

Per prima cosa voglio ringraziare il Professor Stefano Crespi Reghizzi che mi ha seguito in questo percorso, e che con grande passione, interesse e spirito critico ha saputo indirizzarmi verso la soluzione dei problemi che a mano a mano sorgevano. Ringrazio inoltre l'Ing. Simone Campanoni perché senza di lui non sarei mai riuscito ad addentrarmi così in profondità all'interno dell'architettura di ildjit. Ringrazio anche il Professor Giampaolo Agosta, che ha collaborato alla realizzazione di questa tesi sopratutto per quanto riguarda l'aspetto teorico

Un grande ringraziamento particolare va anche ai miei genitori che mi hanno sempre spronato a dare il meglio e che mi hanno supportato e spalleggiato quando, in questi anni, pensavo di non farcela più. Grazie anche a mia sorella che sapeva sempre come tirarmi su il morale cucinandomi i suoi fantastici biscotti che immancabilmente io finivo nel giro di dieci minuti!

Ringrazio inoltre i miei compagni di corso più cari: Vince, Luca, Giangi, Carmi, Enzino e Betta, perché è grazie a loro che questi anni non sono stati solo di studio ma anche di divertimento!

Per ultimi, ma non per questo meno importanti (anzi!), vorrei ringraziare tutti i miei amici che in questo ultimo anno mi sono stati vicino e mi hanno fatto capire cosa vuole dire avere una compagnia, avere degli amici con cui poter ridere, parlare, discutere e anche litigare, ma sempre con affetto; un grazie di cuore ad Albe, Chiara, Pevio, Salva, Alberto ed Ervis! Grazie ragazzi!!

Contents

Estratto in lingua italiana

Questa tesi descrive la progettazione e l'implementazione all'interno di un compilatore dinamico esistente per il byte-code descritto dalla specifica ECMA-335 (meglio noto come bytecode .NET o CIL – Common-Intermediate-Language) di una particolare tecnica di ottimizzazione chiamata *specializzazione*.

Questa tecnica permette di migliorare la velocità di esecuzione di un generico programma sfruttando le conoscenze aggiuntive che possono essere estrapolate dal programma in esecuzione. Queste conoscenze, ad esempio, riguardano quali valori una variabile, un registro o il parametro di una funzione può assumere per la maggior parte del tempo di esecuzione.

Motivazioni

Negli ultimi anni, un crescente interesse è stato rivolto alla compilazione dinamica: tramite questa tecnologia é stato possibile scrivere programmi che, una volta (semi-) compilati in un linguaggio intermedio chiamato bytecode, possono essere eseguiti su qualsiasi macchina, indipendentemente dal tipo di processore e dal sistema operativo.

Tipicamente la compilazione dinamica avviene per mezzo di una virtual machine, un programma che, a tempo di esecuzione, converte il bytecode nel linguaggio macchina specifico dell'architettura presa in considerazione. Dovendo lavorare a run-time è necessario che la traduzione avvenga il piú velocemente possibile, rinunciando spesso ad ottenere un codice migliore.

È comunque possibile, specialmente viste le velocità raggiunte dai moderni

micro-processori, applicare alcune ottimizzazioni standard: dead code elimination, copy propagation, constant propagation, ecc . Queste ottimizzazioni sono inoltre possibili in quanto è sempre più frequente la presenza sul mercato di architetture multi processore, mediante le quali è possibile eseguire il programma su un processore, e la virtual machine, che se possibile esegue una serie di ottimizzazioni, su un diverso processore.

Ovviamente tutte le più comuni tecniche di ottimizzazione disponibili per un compilatore statico possono essere applicate anche al codice generato da un compilatore dinamico. Purtroppo però, se vengono utilizzate senza nessun modifica, non sfruttano appieno tutte le informazioni che possono essere derivate dall'esecuzione dell'ottimizzazione a run time.

In questo contesto si inserisce quindi la specializzazione: tramite questa tecnica è possibile fornire alle altre ottimizzazioni maggiori informazioni che, in questo modo, potranno essere eseguite in modo più aggressivo sul codice, rendendolo migliore e quindi decisamente più veloce.

Specializzazione: una tecnica di ottimizzazione a runtime

Come già accennato, la specializzazione è una tecnica di ottimizzazione che deve essere eseguita a run-time.

Il suo obiettivo è quello di generare diverse versioni di uno stesso metodo originale, sfruttando le informazioni che può derivare a run-time. Queste informazioni riguardano principalmente quali valori una variabile assume per la maggior parte del tempo di esecuzione del programma. Lo stesso discorso può essere fatto per locazioni di memoria, registri e parametri passati ad una procedura.

Esistono diversi tipi di specializzazione a seconda di cosa decidiamo di specializzare. Se si decide di specializzare delle variabili o dei registri, si ottiene la *specializzazione del codice*. Se, al contrario, si vuole specializzare i parametri di

una procedura si ottiene la *specializzazione di un metodo* [1].

Esiste in realtà un terzo tipo di specializzazione chiamato specializzazione dei tipi [2]. In questo caso ciò che si specializza è il tipo di un oggetto che chiama un metodo virtual, cercando in questo modo di ridurre i costi dovuto all'uso di una tabella virtuale.

Questa tecnica è composta da diverse fasi: *pre-processing*, *identificazione degli invarianti*, *analisi dei costi e dei benefici*, *identificazione del codice da specializzare*, *specializzazione del codice*, *dispatching*.

Nella fase di pre-processing viene modificato il codice del programma da generare in modo opportuno, per poter determinare quali sono le variabili (o, allo stesso modo, i parametri di una procedura, le locazioni di memoria o i registri) che assumono per la maggior parte del tempo uno stesso valore e che, per questo motivo, vengono chiamate *statiche*. Una variabile non statica viene invece definita *dinamica*. Solitamente, per eseguire questa fase, ciò che si fa è inserire un profiler all'interno del codice che determina quali sono gli "hot spot" del programma e ne memorizza i valori. Questa fase può avvenire sia off-line (cioè viene eseguito il profiling su dei dati fittizi) oppure on-line, cioè direttamente sul programma in esecuzione con dati reali.

Nella fase di identificazione degli invarianti, vengono analizzati gli "hot spot" generati dalla fase di preprocessing e si va a determinare un insieme di variabili che possono essere considerate statiche. Per ognuna di queste variabili si hanno quindi adesso a disposizione i valori che questa può assumere e la loro frequenza relativa. Questo tipo di analisi può essere eseguita tramite un'analisi di binding-time oppure tramite analisi statistiche sui dati di profiling.

Una volta determinate le variabili statiche è opportuno determinare se i costi per l'esecuzione della procedura di specializzazione sono compensati dal guadagno ottenuto eseguendo il codice specializzato. Esistono diverse tecniche in letteratura per eseguire questo compito: esse si basano principalmente sul guadagno ottenuto

in termini di cicli macchina o di accessi a memoria risparmiati.

Quando la procedura di specializzazione è abbastanza sicura di portare benefici, determina quali punti è necessario specializzare effettivamente. A questo punto viene quindi creato il codice specializzato che andrà a sostituire il codice originale quando necessario. Ad esempio, se si adotta la specializzazione dei metodi, verranno create diverse versioni dello stesso metodo, a seconda dei valori statici noti assunti dai parametri.

Ovviamente non è possibile andare a sostituire indiscriminatamente il codice specializzato all'interno del programma originale: il codice specializzato deve essere eseguito solo se sussistono le condizioni necessarie, cioè se le variabili o i parametri per cui andiamo ad eseguire il metodo specializzato specifico assumono effettivamente questo valore – ciò deve ovviamente essere vero nella maggior parte dei casi, ma dobbiamo essere comunque in grado di eseguire il metodo originale quando il codice specializzato non può essere in nessun caso applicato, in modo da ottenere l'esecuzione corretta del programma. Per fare questo viene inserito un dispatcher, cioè una sorta di costrutto switch che opera nel seguente modo: se i parametri per cui specializzo assumono uno dei loro valori statici allora eseguo il relativo codice specializzato, altrimenti eseguo il codice originale.

Ottimizzazione tramite specializzazione

La specializzazione che si è deciso di sviluppare è una specializzazione dei metodi, quindi la procedura di specializzazione è interessata a determinare i valori statici dei parametri delle funzioni specializzabili. Per costruire una siffatta procedura nella pratica, si è deciso di seguire a grandi linee i passi descritti nel paragrafo precedente. I nuovi passi per eseguire la specializzazione sono i seguenti: preprocessing, profiling, specializzazione del metodo chiamato e inserimento del dispatcher all'interno del chiamante.

La prima fase, quella di preprocessing, deve essere eseguita off-line. Ciò che si

richiede è che un tool esterno indichi alla procedura di specializzazione quali sono i metodi che è opportuno specializzare. Questa procedura deve inoltre descrivere come è possibile specializzare i metodi indicati, definendo quali parametri possono essere considerati statici per ciascun metodo. Ciò può essere fatto sia da un generico tool esterno, che analizza il byte-code CIL e genera un file di descrizione, sia da un compilatore "amico" che aggiunge dei metadati alla descrizione del metodo stesso.

La fase di profiling ha l'obiettivo di determinare se effettivamente è utile specializzare i metodi definiti dalla fase precedente, e, in questo caso, quali sono i valori statici da assegnare ai parametri. Per fare ciò, questa fase fa uso di due distinti profiler. Un primo profiler, leggero e veloce, ha lo scopo di verificare che il metodo viene effettivamente eseguito un numero sufficiente di volte. Il secondo profiler, al contrario molto più pesante, ha lo scopo di immagazzinare in memoria i valori dei parametri attuali dei metodi da specializzare una volta che, tramite il primo profiler, si è giunti alla conclusione che il metodo è stato chiamato molte volte, e quindi anche in futuro è possibile che venga richiamato spesso. Si è deciso di suddividere questa fase tramite due profiler proprio perché, essendo il secondo profiler abbastanza pesante e visto che deve fare molti accessi a memoria, è necessario verificare preventivamente che il metodo venga chiamato un numero sufficiente di volte, in modo da non appesantire inutilmente l'esecuzione a run-time del programma, peggiorandone le prestazioni.

Sempre all'interno della fase di profiling, una volta che il secondo profiler ha collezionato abbastanza informazioni, è possibile determinare quali sono i valori statici dei parametri da specializzare con una semplice analisi delle frequenze dei valori. Se non viene trovato nessun valore statico per i parametri, allora si inserisce nuovamente all'interno del codice il primo profiler; se invece vengono trovati dei valori si passa alla fase successiva di creazione dei metodi specializzati.

Esiste in realtà anche un terzo profiler che ha lo scopo, per le chiamate virtuali,

di stabilire qual è il tipo dell'oggetto che chiama effettivamente il metodo. Questo profiler assomiglia molto al secondo profiler appena descritto per modalità di funzionamento. Una volta che per le chiamate virtuali specializzabili sono stati trovati i tipi degli oggetti che maggiormente vengono utilizzati, è possibile sostituire la chiamata virtuale con un dispatcher che verifica se il tipo attuale dell'oggetto su cui viene chiamato il metodo coincide con uno di quelli precedentemente trovati. Se questo avviene, si inserisce una chiamata diretta al metodo in questione, senza passare per una (costosa) chiamata virtuale che fa uso di una tabella di look-up. Se ciò, al contrario, non avviene, allora deve essere eseguita l'originale chiamata virtuale. Ovviamente, le chiamate ai metodi non virtuali inserite tramite questo tipo di dispatcher di tipo possono essere a loro volta specializzate.

Una volta che il secondo profiler ha terminato il proprio compito e siamo in possesso dei valori statici dei parametri dei metodi, è possibile creare una versione specializzata del metodo stesso. Di questo nuovo metodo specializzato è quindi necessario modificare la signature eliminando quei parametri che vengono specializzati. Una volta fatto ciò si può semplicemente copiare il contenuto del metodo originale, ed inserire, all'inizio del codice, degli assegnamenti che sostituiscono i parametri in modo che assumano i valori statici precedentemente determinati. In questo modo, essendo a conoscenza dei valori effettivi dei parametri, il compilatore può applicare al metodo un'ottimizzazione molto più aggressiva ed efficace.

Quando siamo in possesso di tutti i metodi specializzati a cui siamo interessati, è necessario inserire il dispatcher che funziona allo stesso modo in cui è stato definito nella sezione precedente.

Ad un certo punto della propria esecuzione, il programma su cui abbiamo applicato la specializzazione potrebbe cambiare completamente il proprio comportamento e i valori statici precedentemente trovati potrebbero non corrispondere più ai valori statici attuali che il programma presenta. Per questo motivo è necessario de-specializzare, cioè bisogna eliminare il dispatcher precedentemente inserito e

fare ripartire la procedura di specializzazione dalla fase di profiling.

Alcuni possibili problemi non sono comunque stati presi in considerazione a causa della loro complessità. Ad esempio, un primo problema nasce appunto dal fatto che un programma può cambiare il proprio comportamento in modo del tutto arbitrario in qualsiasi momento. La procedura di specializzazione dovrebbe quindi cercare di essere il più reattiva possibile nel determinare questa avvenimento. Per fare ciò esistono diverse tecniche. Ad esempio, all'interno del dispatcher, è possibile contare quante volte viene chiamato il metodo originale e quante volte uno dei metodi specializzati. È necessario però riuscire a determinare in modo corretto i parametri della procedura di specializzazione in modo tale che il cambio di fase del programma venga determinato il prima possibile, cioè occorre settare in modo corretto, ad esempio, la soglia oltre la quali si considera che sono state eseguite troppe chiamate al metodo originale rispetto al metodo specializzato.

Implementazione del sistema

Per implementare il sistema descritto nella sezione precedente si è deciso di modificare la pipeline di compilazione del compilatore ILDJIT, un compilatore open-source sviluppato internamente al Politecnico di Milano. Questo compilatore è stato scelto perché fornisce una vasta gamma di ottimizzazioni utili ai fini della specializzazione e, in particolare, permette di eseguire la constant propagation, copy propagation, dead code elimination, strenght reduction, ecc .

Prima di eseguire la fase di specializzazione vera e propria, durante la fase di bootstrap ed inizializzazione del compilatore, vengono innanzitutto letti da un apposito file quali metodi possono venire specializzati e come.

Per poter eseguire la fase di specializzazione è stata modificata la fase di traduzione dal linguaggio intermedio (IR) al linguaggio macchina (JIT) del compilatore: ogni volta che viene determinata un'istruzione di chiamata a funzione (anche virtuale) che può essere specializzata, viene modificato il codice relativo a

quella chiamata a seconda dello stato attuale in cui si trova, cioè se il call-site si trova nello stato di profiling o di dispatching.

Per implementare il primo profiler si è deciso di inserire all'interno del codice della funzione chiamante un semplice contatore che viene incrementato ogni volta che il flusso di esecuzione raggiunge l'istruzione di call. Quando il contatore raggiunge una determinata soglia, vengono rimosse le istruzioni di aggiornamento del contatore e si richiede al thread di compilazione di ricompilare il codice del chiamante in modo che possa inserire il secondo profiler.

Il secondo profiler, per poter memorizzare i valori dei parametri attuali della procedura da specializzare, alloca della memoria in cui poter memorizzare tali valori. Per fare questo inserisce all'interno del codice delle opportune istruzioni che accedono alla memoria, rendendo di fatto il secondo profiler molto più "pesante" del precedente. Una volta che la memoria in cui vengono scritti i valori dei parametri è stata completamente riempita, il secondo profiler fa un analisi di frequenza dei valori assunti da ciascun parametro. Se un parametro assume un certo valore con una frequenza che supera una determinata soglia allora si considera il valore statico per quel parametro.

A questo punto, ottenuti i valori statici per i parametri del metodo da specializzare, viene creato un nuovo metodo clonando il precedente e specializzandolo come definito nel precedente paragrafo. Si richiede inoltre al thread di compilazione di ricompilare il metodo chiamante per eliminare le istruzioni aggiuntive inserite dal primo profiler ed inserire le istruzioni relative al dispatcher.

Una volta ottenuti tutti i metodi necessari alla specializzazione, viene inserito il dispatcher a cui viene aggiunto un contatore per verificare che, per la maggior parte delle volte, venga chiamato uno dei metodi specializzati. Se ciò non avviene si richiede una nuova compilazione del metodo chiamate che elimina il dispatcher e inserisce nuovamente il primo profiler.

Per rendere più semplice l'implementazione del sistema si è deciso di "fondere"

il primo profiler con il profiler dei tipi per le chiamate virtuali – anche chiamato profiler zero. Per questo motivo, ogni volta che la procedura di specializzazione determina una chiamata virtuale inserisce sia del codice per il primo profiler, sia del codice per salvare in memoria i tipi attuali degli oggetti che richiamano il metodo. Tutto ciò può essere fatto seguendo lo stesso schema del secondo profiler.

Risultati Sperimentali

I risultati sperimentali ottenuti mostrano che la tecnica di specializzazione adottata in concomitanza con le altre classiche ottimizzazioni presenti fornite da `ildjit` permette di ottenere miglioramenti nel tempo di esecuzione compresi tra il 2% e il 38%. Questi valori sono compatibili con quelli riportati dalla letteratura classica della specializzazione [3, 4]. D'altro canto l'uso della specializzazione deve essere opportunamente supportato da un insieme di informazioni che, al momento, possono essere generate solamente analizzando manualmente il codice, e quindi potrebbero non essere in grado di determinare suggerimenti di specializzazione più opportuni.

Conclusioni e sviluppi futuri

I risultati ottenuti mostrano come, dopo aver estrapolato opportune informazioni dal codice, sia possibile effettuare la specializzazione sul codice in modo tale da ottenere dei risultati significativi e dei miglioramenti tra il 2% e il 38%. Il maggiore contributo dato da questa tesi allo studio della specializzazione riguarda l'utilizzo di serie di profiler inseriti nel codice a differenza di quanto fatto fino ad ora dove se ne è utilizzato sempre solo uno. Inoltre, la tecnica proposta, ad eccezione della fase di preprocessing, viene eseguita completamente on-line, mentre tipicamente in letteratura la fase di profiling viene eseguita a off-line, portando quindi la procedura di specializzazione a lavorare su dati che non necessariamente saranno quelli effettivi con cui ci si deve confrontare quando il codice viene compilato ed

eseguito a runtime.

I risultati ottenuti sono infine incoraggianti per lo sviluppo futuro di questa tecnica all'interno di un compilatore di dinamico. Successivi sviluppi possono riguardare l'implementazione e lo studio di tecniche più fini di selezione metodi da specializzare e l'utilizzo di tecniche avanzate, qui solo accennate, quale ad esempio il *window specialization growing*.

Organizzazione dei capitoli

La tesi è organizzata come segue.

Il capitolo 2 fornisce una panoramica esaustiva sullo stato dell'arte attuale della tecnica di specializzazione. Verranno inoltre descritti alcuni tool e compilatori esistenti che operano utilizzando una fase di specializzazione.

Il capitolo 3 presenta una descrizione della compilazione dinamica, con particolare riferimento alla piattaforma .NET e il suo linguaggio bytecode chiamato CIL, definito dallo standard ECMA-335. Verrà infine presentato anche il compilatore ILDJIT su cui si è deciso di implementare la specializzazione.

Il capitolo 4 descrive una possibile generica tecnica di specializzazione. La descrizione è molto ad alto livello e non fa specifico riferimento a nessun compilatore e a nessun linguaggio in particolare.

Il capitolo 5 mostra come sia stata effettivamente implementata la specializzazione all'interno di ILDJIT.

Il capitolo 6 mostra i risultati ottenuti tramite l'esecuzione dei benchmarks eseguiti sul codice quando a questo vengono applicati tre diversi livelli di ottimizzazione: livello 0 (senza ottimizzazioni), livello 3 (con ottimizzazioni tipiche di ILDJIT) e livello 5 (con la specializzazione dei metodi).

Il capitolo 7 espone le conclusioni che sono state tratte da questa esperienza e delinea i possibili sviluppi futuri.

Le appendici A e B mostrano i valori numerici ottenuti dall'esecuzione dei

programmi di benchmark e i parametri con cui è stata eseguita la specializzazione su tali programmi.

Summary

In this thesis I am going to describe how I have defined and implemented a way to support the partial evaluation process inside ILDJIT, a Just-In-Time compiler developed internally at Politecnico di Milano.

Dynamic compilers, and in particular ILDJIT, can obtain great benefits with the introduction of this technique because it can enable a much higher possibility to exploit other available optimizations.

My work – adapting an existing JIT compiler in order to insert a specialization phase inside the compilation process – allows to perform experimental studies on partial evaluation techniques and how they can increase the execution speed of the running program.

Experimental runs of benchmarks, though initial, confirm the effectiveness of dynamic compilation for CIL bytecode and provide initial indications on the speed-up obtainable by adding a specialization phase inside a dynamic compiler.

Chapter 1

Introduction

Nowadays, one of the most investigated fields of research about formal and programming languages and compilers concerns the ability of these languages and compilers to generate portable code, i.e. code that can be executed on different machines equipped with totally incomparable processing units and operating systems.

This problem has been solved with the introduction of virtual machines. A programmer can write a program with his favourite program language and, with the use of a static compiler, it can generate the same program written in bytecode language, that is intrinsically portable. To execute this bytecode, there is anyway the need of a second step that translates the bytecode to an executable program. This final process is performed by a *virtual machine*. The generated executable program is now machine dependent and should use all possible features of the processor that will execute the code. For example, if the CPU supports MMX instruction set, then the virtual machine should generate code using, where possible, also this kind of instructions in order to increase the execution speed of the program.

A virtual machine can translate at run-time bytecode into an executable form substantially in two ways: the bytecode can be interpreted or actually recompiled inside the target executable language. For our purpose we will consider this last case.

Obviously, since the compilation process takes place at run time, it should be as fast as possible, possibly leading in order to a poor code generation to reduce the compilation over-head. Anyway, though at run-time, a lot of standard optimizations are possible, for example standard copy propagation, constant propagation, dead code elimination and strength reduction can be applied as well. Since these optimizations are executed at run time, they can benefit of the fact they can be able to exploit more information than what a static compiler can do, because they can now determinate which actual values variables, registers and functions parameters assume.

This thesis dissertation is based just on these facts. Since the dynamic compiler can possibly know which values a variable, a register or a function parameter can assume in practice, it can use this information to apply a more aggressive optimization on programs. Moreover, once we know these values for actual method parameters, the compiler can generate different versions of the same method, each one optimized in different ways that depend on the values assumed by the parameters. This way to proceed is called *partial evaluation*, also known as *specialization*.

Therefore, the aim of this thesis is to modify the compilation pipeline of a given dynamic compiler in order to exploit run-time information and apply the partial evaluation optimization technique. This would increase the speed-up obtainable by the optimizations executed by the dynamic compiler.

Summarizing, the project is composed of the following main phases:

- design of a generic application of partial evaluation in a dynamic compiler, in order to exploit dynamic information available;
- design and implementation of the technique previously defined inside the ILDJIT compiler, modifying its compilation pipeline;
- performance evaluation of the implemented techniques through experimental results;

Experimental results are encouraging, since they are comparable – in terms of maximum speedup achieved on execution time – with those reported in the literature [3, 4].

As expected, the executed benchmark shows that the code generated using the described optimization runs faster only if the specialized code is executed a sufficient number of times, and if the determination of which methods are eligible to be optimized by the partial evaluation process is done very carefully.

The rest of the thesis is organized as follows.

Chapter 2 gives an overview of the partial evaluation optimization techniques. It introduces the basic concept of specialization and describes in detail how it can performed. In this chapter I will also present a short list of some available tools that already perform, with some limitations, the specialization of the code inside both dynamic and static compilers.

Chapter 3 presents a description of the state of the art for Dynamic Compilation, with particular emphasis on .NET architecture and the .NET bytecode (more formally the CIL language described by the standard ECMA-335). Here, the existing execution engines will be presented: interpretation, just-in-time (or dynamic) compilation and Ahead-of-time compilation. Moreover, this chapter describes the ILDJIT platform used inside this thesis, and presents others available virtual machines that support the execution of CIL bytecode.

Chapter 4 describes a possible generic implementation of partial evaluation. This description does not depend on the particular architecture adopted to execute the code. For this reason it could theoretically be applied to every virtual machine, regardless the language or the architecture used.

Chapter 5 describes my implementation of the techniques and choices for the implementation the partial evaluation process inside the ILDJIT dynamic compiler. Here I shall also present a simple description of the internal architecture and execution model of the compiler that I have chosen to use.

Chapter 6 shows the experiments that have been performed on the JIT compiler, comparing the execution time of a suite of benchmarks executed with three different levels of optimization: without any optimization (level 0), with standard optimizations provided by the compiler itself (level 3) and, finally, with the partial evaluation optimization technique active (level 5).

Chapter 7 exposes the conclusions that have been collected from this experience, and focuses the attention on particular issues that might outline the direction for future works.

Appendixes A and B list the numerical values obtained from a subset of the executed benchmarks and the specialization parameters used to run such programs.

Chapter 2

State of the art for partial evaluation

Program specialization, also known as *partial evaluation*[1], with respect to run-time invariants, is an optimization technique that has shown to be able to improve substantially the performance of programs: it allows a program to adapt to execution contexts that are valid for a limited time or for a single program execution run-time context.

Run-time specialization has been actively investigated in a variety of areas: operating systems, generic purpose library, dynamic compiler/interpreter and CPU-bound algorithms like those used for Digital Signal Processing (DSP). For example, in the context of file system operations (used, for instance, in a library), run-time invariants can include the type and the opening-mode of a file and the device where it resides. When the file is being opened, at run-time, obviously, invariants become available and can be used to specialize read and/or write procedures: this specialization eliminates redundant code, like controls on the file type or the device used. Removing all these checks (and hopefully a lot of unused code) and performing other optimization on specialized code (dead-code elimination, constant propagation, loop unrolling, ...) the specialized program should yields significant improvements and therefore run faster than the unmodified one.

[1]Some authors consider specialization a particular way to implement partial evaluation, for our purpose they are basically the same thing.

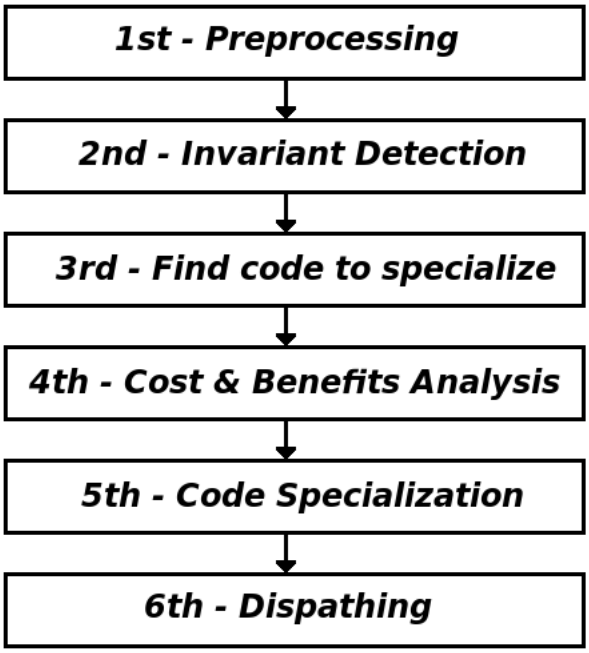

Figure 2.1: Program specialization process

Depending on the specialized program you can theoretically obtain speedups from 2x to 40x in program time execution.

2.1 Phases of program specialization

Program specialization optimizing process can be broadly divided in these phases as shown in figure 2.1: preprocessing, invariant detection, benefit and cost evaluation, detection of the code that has to be specialized, code specialization and dispatching.

2.1.1 Preprocessing

The first phase of program specialization is the *preprocessing*. During this part of the process the *specializer*, i.e. the procedure that performs partial evaluation, instruments the given program in order to be able to obtain profiling information useful for the next phases. Instrumentation can be performed either at source file level (inserting specific function call in a C or Java source file for instance)

or at assembler/bytecode level. Profiling can be used to collect different kind of information.

Most specializers use profiling to have some information about actual parameters passed to functions during calls at run time (as described in [5]); this information is very useful if you are going to perform method specialization (see paragraph 2.2.3).

Other specializers rely on a different kind of profiling: they need information about which values are assumed by a variable (placed either in a register or in memory) in most cases. For this reason, they could place profiler calls at any point in the code, while the first profiler method puts profiling calls only near a method call. Therefore, this way of operating could be much more expansive than the previous one and so profiling calls should be inserted only where there is an higher likelihood that the variable, which you want to specialize, keeps almost always the same value. This probability estimation could be done a priori using some cost and benefit evaluations as described in [6, 4] and that we will explain next in this chapter. This last kind of profiling method is useful when you want to specialize parts of a method (called *trace*[2] in this context).

A brand new way to profile a program has been recently investigated. This way of operating is called *shadow profiling* [7]. This kind of profiling tries to avoid the costs, creating a "shadow copy" of instructions you want to profile simply by doing a fork and executing both copies of the method (the original one and the shadow one) at the same time, possibly on two different processors. Anyway, even if this way of proceeding could avoid the costs of profiling, it is quite complex to implement because it should take care of a large number of problems: for example it should prevent the shadow copy to write on the same files or shared memory of the original copy of the programs. For these reasons, shadow profiling should be used only when strictly needed, i.e. when the profiling is too much costing respect

[2]In compiler literature a trace is a sequence of basic blocks, in this context a trace is just a sequence of instructions.

to the gain given by other optimizations that rely on that profile data.

The result of this phase is a set of raw profiling data that must be examined in order to understand when and where it is useful to apply the specialization.

2.1.2 Invariant detection

The second phase of the program specialization is the *invariant detection*.

In this part of the process the specializer analyzes the data obtained from the previous phase and tries to detect which actual parameters of a method (or in the same way variables, registers or memory locations) are *run time invariants*, i.e. during the program execution they take only few values most of the time, although they could theoretically assume a large range of values.

In this context we say that such an invariant parameter is *constant like* and we can state that its value is known at *specialization time*, i.e. when the specialization process takes place. Likewise we say that a parameter is *non-constant like* if it's not known at specialization time but only at run time[3]. Actually a variable cannot always be defined as constant like or non-constant like because this distinction is context dependent: the same variable could be constant like in a part of the method and non-constant like in another one. So, distinguishing between constant like and not-constant like variables is not so easy as it could seem.

There are basically two techniques to determinate weather a parameter is constant like or non-constant like: the first one uses the binding-time analysis (as explained in [1], [8] and [9]) and the second one uses an heuristic that exploits some statistical analysis on profiling data (as shown in [6] and [5]). We are going to talk about these two techniques next in this chapter (see 2.2.4 section).

[3]In literature sometime are used the terms *static* and *dynamic* in this context instead of *constant like* and *non-constant like*, but this can be quite misleading, since the same terminology is used to distinguishing between static-compile time and dynamic compile time.

2.1.3 Cost and benefit analysis

In the third phase of the program specialization we evaluate *costs and benefits* that arise using specialization. Actually, this phase should be kept into account also in the previous one so these two phases can be merged together, but for clarity and simplicity I have preferred to keep these two key aspects separated.

The maximum cost that we should take into account is the time spent to execute specialization. This cost should be kept into account only if you perform specialization at run time: in this case we say that the time spent for specialization should be amortized on the whole time spent for executing the specialized code.

Another point of cost in this optimization is that often specialized programs require much more memory and have to generate much more line of code than unspecialized ones.

The last cost parameter that we could consider is the time spent for recompilation in a Just In Time compiler. For these compilers, each time a new specialized method is created, it must be compiled again subtracting time to the real execution of the program. This cost could be significantly reduced if the JIT compiler is able to work on different processors using one thread for compilation and another one for the method execution.

A JIT compiler that uses these techniques is the ILDJIT compiler described in chapter 3. Now the benefits that this optimization could give to a program will be presented. First of all, a single specialized method is very often smaller than the unspecialized one because some code could be evaluated totally or partially at specialization time, so these instructions could be removed from the source. Because the specialized code is smaller, it should run faster and could be easily used for inlining optimization. Actually, the problem at this level is to estimate both costs and benefits because real ones could be known only at run time. There are several ways to evaluate a priori these two key aspects; we will discuss the most interesting one in section 2.2.5.

2.1.4 Code to specialize detection

The next step, in the program specialization process, uses information from previous phases to detect which method (or more generally which parts of a method) we should specialize in order to gain speed up in program execution.

The *selection of what to specialize* depends on the choices made in the previous phases.

For example, if we have chosen to specialize a method using profiling information we could use the cost and benefit evaluation to select the method where the ration $cost/benefit$ is lower than a given threshold.

Otherwise, always for method specialization, we can follow the method selection presented in [5, 6]. In a similar way, we can select parts of a single method (traces) following the rules described in [1, 4, 10].

2.1.5 Code specialization

The fifth phase is the simplest of the specializing process and, eventually, it *builds the specialized code*. First of all we have to know the run time invariants detected in the second phase; supposing that we have discovered that a parameter of a function assumes only two values in most cases, we can now create two different specialized method for each value.

The simplest way we can operate is just to assign the known value to the parameter and rely on other optimizations like constant propagation, constant evaluation and other discussed in section 2.3. This way of operating works only if you know at specialization time all the possible values of constant like parameters.

However, in a JIT compiler this is not true, so the specializer must work incrementally because it knows values for constant like parameter little by little and we usually create one specialized method at time when a new value for a constant like parameter/register/variable is known.

Examples of how a specialized method can be built are to be found in [1, 9].

Listing 2.1: Method to specialize

```
int power(int a, int b){
   int i = 0;
   int result = 1;
   for(i = 0; i < b; i++)
      result = result * a;
   return result;
}
```

2.1.6 Dispatching

The target of the last phase is to find a way to call specialized method when needed and to call the original one when requested values do not fit: this process takes the name of *dispatching*. The simplest way of proceeding is injecting a sequence of `if` statements that selects either one of the specialized versions or the original method in the code that has to be generated, simply by checking that the values of actual parameters are equals to the ones that the specializer selected for partial evaluation. Obviously all these comparisons are not free and, in the cost and benefit analysis, the specializer should take into account also this time consumption because it could be very expansive to generate too many specialized methods.

Moreover, equals comparison could be very slow if you try to compare two data structures (like C `struct` or Java objects) or two float or double variables. Regardless of how the dispatcher is executed by the specializer, it can be inserted in different places: in the compiler itself (if it's a JIT compiler), in the generated specialized source code or in the generated specialized assembler/bytecode. Examples of dispatching method can be found in [1, 9].

An explanatory example

We are now going to have a look to a simple example that shows how partial evaluation can improve the execution speed-up. Suppose that the specializer has

Listing 2.2: Method specialized for b=3 and b=4

```
int power_3(int a){
   int i = 0;
   int result = 1;
   for(i = 0; i < 3; i++)
      result = result * a;
   return result;
}

int power_4(int a){
   int i = 0;
   int result = 1;
   for(i = 0; i < 4; i++)
      result = result * a;
   return result;
}
```

Listing 2.3: Method specialized and optimized for b=3 and b=4

```
int power_3(int a){
   return a*a*a;
}

int power_4(int a){
   return a*a*a*a;
}
```

already executed the first four steps of the specialization process and that we want to specialize the method in listing 2.1, knowing that parameter `b` is constant like and that it assumes in most cases value 3 or 4.

As previously explained, we have to create other two versions (specializations) of the same function, one for each value assumed as more likely at specialization time. You can see the result for both value 3 and 4 in listing 2.2.

These new two versions are not so different from the original one at this time: this is because specializer has not applied any other optimization but only constant propagation. If we apply other optimizations, like loop unrolling and common sub-expression elimination, we can obtain new methods like those shown in listing 2.3

Listing 2.4: Method dispatcher for b=3 and b=4

```
int power_dispatcher(int a, int b){
  switch(b){
      case 3:
          return power_3(a);
          break;
      case 4:
          return power_4(a);
          break;
      default:
          return power(a,b);
  }
}
```

Listing 2.5: Better method dispatcher for b=3 and b=4

```
int power_dispatcher(int a, int b){
  switch(b){
      case 3:
          return a*a*a;
          break;
      case 4:
          return a*a*a*a;
          break;
      default:
          return power(a,b);
  }
}
```

that are obviously much more efficient than respectively previous versions.

At this point we have two specialized versions of the method and we have to create a dispatcher that should be substituted wherever there is a call to the original function. An example of dispatcher is shown in listing 2.4.

The last algorithm shown can be improved further: we can use inlining to save a call to `power_3` and `power_4` obtaining a new better dispatcher shown in listing 2.5.

Therefore, the created method should run faster if in most cases `power` is run with `b=3` or `b=4` because it avoids a lot of `jump` used in the original method

generated by the `for` statement. Another way to enhance performance is applying again the inlining optimization for the dispatcher method: in doing so you can save a function call in most cases, increasing performance because calls are often very expensive to execute because they require a context switch.

2.2 Program specialization in practice

In the previous section how partial evaluation can be achieved has been explained, describing each single passage necessary in order to obtain this optimization.

In this chapter some specific partial evaluation techniques will discussed in more detail, highlighting for each one the advantages and disadvantages referring in particular to just in time compilers.

2.2.1 On line and offline execution pattern

Once you have decided how to implement all phases, you have to select when executing them: they can be run *online* or *offline*. Online means that the step is executed at run time (just like in a JIT compiler) while offline means that the phase has been executed at compile time.

Both options have advantages and disadvantages: online is very useful because you could work on actual data but it may lead to a performance degradation in program execution. On the contrary, if you choose offline, you have not to worry about time consumption but you could only work on profiling data that could be very different from the real one. These two ways of implementing partial evaluation can work together: you could for example execute the first three phases of program specialization offline, while you can build specialized method at run time relying on actual values for constant like parameters.

2.2.2 Dynamic compilation for specialized method

In the previous sections we have already mentioned that partial evaluation is a process that can be done both at run time and at compile-time.

Suppose now that the specializer is working in a just in time compiler. Regardless which specialization method you choose for the implementation, once you have defined a new specialized method, it is very likely that you have to recompile it, except when it works directly on assembler code. When the specializer needs to recompile the specialized method it has two choices: recompiling from original source code (if the specializer generates high level source code, for example when it generates C code) or recompiling from bytecode into target assembler machine code. It is pretty obvious that recompile from source code is more expansive than recompile from bytecode, also because in this way you often have to run an external compiler.

Unfortunately, generating directly assembler or bytecode is not always feasible because these two languages could be too much complicated and/or machine dependent. For this reason, in JIT compilers, the optimal choice is to generate a specialized method in the intermediate representation language used by the compiler. This language is often easier to manage than bytecode/assembler but, at the same time, faster to compile than high-level source code, so this is the optimal choice for JIT compiler.

2.2.3 Method specialization, code specialization and type specialization

The program specialization can be performed either at method level or at instruction level as it has been said before. In the first case the target is to generate specialized method while in the second one you can generate specialized code only for a part of method, that in this context is called *trace*. An example of *method specialization* has been shown in the previous section and this is the most com-

mon specialization technique used. This way of operating is very useful if you are going to specialize a whole library, because you can remove unused code and useless check at specialization-time and at run-time, obtaining a smaller and faster code. Actually what you are doing in this case is called *component specialization* [11, 12] but in the end it is a specialization on all methods (components) of a library. Performing method specialization is quite easier: you just have to understand which function parameters are constant like and which are non-constant like and specialize on the constant like one; the most difficult part is understanding if a parameter is constant like or non-constant like.

Much more complex is the case where you are going to perform generic *code specialization*. The trace to specialize can start and end at any point inside a method, for this reason you have to use more sophisticated cost and benefit analysis which can understand, in a deeper detail, the influence of an instruction on the others, if it is specialized. However, this analysis must run very fast if we are executing specialization in a JIT compiler. It is quite obvious that method specialization is a particular way of implementing code specialization: if you select the trace that cover all instructions inside a method you can realize a method specialization using code specialization.

Not only methods and generic code blocks can be specialized, but also the dynamic type of the object that calls a method [2, 13] (actually there is a *virtual call* to the *virtual method*). In this context, if the specializer finds which are the most frequent dynamic types of the object that calls the virtual method we can partially avoid the cost of the virtual dispatching table used to detect which method has be actually called, depending on the dynamic object type.

2.2.4 Binding-time analysis and value profiling

In this section some techniques described in partial evaluation literature are going to be explained. Some of these methods use *binding-time analysis* while others use *value profiling*.

The aim of binding-time analysis process is to annotate the code in order to understand which instructions and variables are constant like, which ones are non-constant like and which could be evaluated partially at specialization time. This method can work without the need of particular instrumentation: in most cases the starting input to this evaluation is given directly from the user (i.e. the developer) that decides for each formal parameter of a method if it is constant like or not-constant like. However, there are automatic tools that could perform this decision using, for example, profiling information.

Value profiling, instead, needs to be executed after the instrumentation of the program. As already said, you can place profiling calls everywhere in the program being careful to not exaggerate especially if you are working in a JIT compiler. The purpose of this method is to understand which values a variable/register/parameter assumes in most cases.

Anyway, regardless of which method the specializer decides to use, it is important to understand that, within this process of analysis, it just wants to understand where specialization can be applied whatever partial evaluation will be applied or not: this choice will be done during cost and benefit analysis. In order to better explain these different techniques, in the next paragraph it is possible to find some examples that will give an idea on how both methods can be executed in different context.

Binding-time analysis for C language

Binding-time analysis can be executed directly on C source code [1]. Consider the code example shown in listing 2.6 extracted from the same article previously mentioned.

On the left you can find the original function, while on the right you can see the same function but decorated with four different tag: *id*, *ev*, *red* and *reb* that are used by the specializer to perform partial evaluation. First of all consider the two function parameters and suppose that someone (for example the programmer

Listing 2.6: C source code before and after binding-time analysis from [1]

```
int f(int x, int y){                    int f(int x^ev, int y^id){
    int l;                                  int l;
    l = 2*x;                                (l = 2*x)^ev ;^red
    if(l == 2)                              if^red (l==2)^ev
        l = l + y;                              l^id =^reb l^ev +^reb y^id ;^reb
    else l = y * x;                         else l^id =^reb y^id *^reb x^ev ;^reb
    return l;                               (return l;)^id
}                                       }
```

or some automatic analysis tool) has told us that `x` is constant like and `y` non-constant like. For this reason `x` has been labelled as *ev*, which means *evaluate*, because this parameter is known at specialize time and so can be evaluated. Vice versa `y` has been labeled as *id*, that means *identifier*, because its value is unknown and in the specialized code the specializer must keep this variable as identifier.

Now it is possible to start analyzing the instructions of the function. First of all, instruction `l = 2*x` has been marked as evaluable because on the right side of the assignment there are only evaluable variables so the whole assignment can be evaluated by the specializer at compile time. The trailing ";" has been marked as *red* which stands for *reduce*: as previously said, the instruction `l = 2*x` will be executed by the specializer so it can safely be "removed" (reduced) from the function source code. Likewise, the conditional statement can be reduced and so can the test expression over the `l` variable because it has been previously marked as evaluable. Still the branches of the `if` statement have been marked as *reb*, which stands for *rebuild*: in these statements there is something constant like and something else non-constant like thus after the assignment `l` will be labeled with *id*. Moreover, the instruction must be kept "as it is" assigning only at specialize time the known constant like values, like the `l` variable on the right side of the assignment.

At this point we can have a very precise idea of which instructions could be specialized and the specializer can infer the relation between specialized variable

and benefits. The main drawback of this way to operate is that someone has to give a "starting point" to the specializer. This "starting point" can be found by an automatic tool (for example Calpa [8]) using profiling data but, if you use such information, it would be better do not spend other time with binding-time analysis especially when working in a JIT compiler.

Binding-time analysis for bytecode languages

Binding-time analysis can be executed directly on an intermediate low level language (like Java Bytecode [9]). Actually when working directly on java bytecode it could be very complicated to discriminate between constant like and non-constant like instructions because this is a stack based language so each instruction must keep into account if the value on the top of the stack is either constant like or non-constant like. For this reason an instruction can be executed sometimes as non-constant like and other times as constant like. On the contrary, if the bytecode language is not stack based but works on machine registers[4], you can use the same technique explained for binding-time analysis on the C language.

Value profiling for method specialization

Value profiling has been deeply studied for method specialization. The basic idea is instrumenting the code in order to obtain a list of performed calls with their associated actual parameters. Once the specializer owns this list it can try to understand if a parameter is constant like or non-constant like. Here the approach described in [5] will be followed.

Method value profiling is performed at procedure level: each procedure call is profiled, because it can be called in many different places with different argument values. One of the difficulties in the value profiling occurs when the argument size is dynamic (for example this happens when you are using the classic C `printf` function). Another difficulty arises when the arguments have complex data type

[4]For example the intermediate language IR for the ILDJIT compiler works on registers.

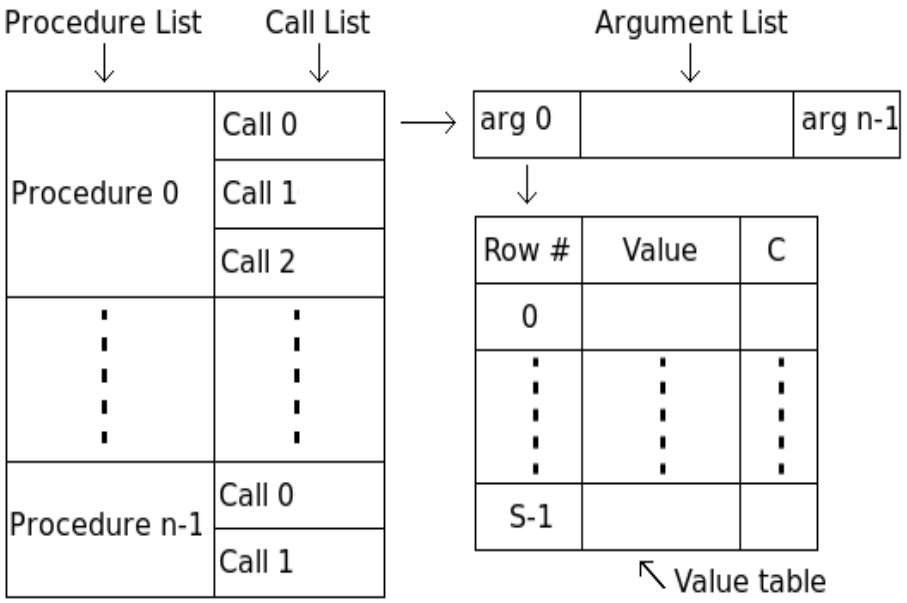

Figure 2.2: Internal data structure of method value profiling

because complex data type require hierarchical traversal for value profiling. For this reason, profiling is restricted to elementary data type scalar and array variables. When a procedure has both complex and elementary data type parameters, the choice is excluding complex variable from analysis and consider them as always non-constant like. In this approach pointers to procedures are not considered due to their dynamic nature. Figure 2.2 shows the internal data structure used by the value profiling system. Each procedure has a list of procedure calls that can be in different places of the code (formally this places are called *call site*). Each procedure call in the list has a list of arguments and each argument in this list satisfies the type constraints mentioned above and has its own fixed size value table to record the values observed and their frequency. Each row in the value table consists of three fields: index field, value field and count (`C`) field. The index field represents not only the index of the row, but also the chronological order of the row in terms of the update time relative to other rows. Thus, the larger the index is, the more recently the corresponding row is updated. The value field is used to store the observed value, and the count field of each row counts the num-

ber of observations of the corresponding value: the table is continuously update whenever the corresponding procedure call is executed.

At the end of the profiling, each parameter of the method is examined to find those values which are frequently observed and only the argument-value pairs, which satisfy user defined constraint called `OT` (*Observed Threshold*), are reported to the user. For this purpose, OR_i is calculated for each row r_i in the value table as follows:

$$OR_i = \frac{c_i}{f}$$

where c_i is the row counter and f is the visiting frequency of this call site. The larger OR_i is, the more frequently the value is observed. When OR_i is smaller than OT, the value in row r_i is disregarded.

As mentioned above, the size of the value table is fixed to save memory and update time, so we have to define a replacement rule policy. Suppose that initially c_i of each value table is 0. Thus if a new value is observed and at least one of c_i is 0, the new value is recoded in r_i which has the smallest index among these rows. On the other hand, when the table is full (there is no c_i which is 0), the following formula is used to select the row which has to be replaced

$$rf_i = W * i + (1 - W) * c_i$$

where rf_i is the *replacement factor* which is the metric to decide which row is to be replaced. The smaller rf_i is, the more likely r_i will be selected for replacement. The weighting factor W is used to specify the importance of the chronological order relating to the observed count c_i. The row r_i which has the smallest rf_i is deleted from the table and substituted with the new value.

After this profiling has been executed the specializer can select which result can give better results in partial evaluation following for example the rules exposed in 2.2.5.

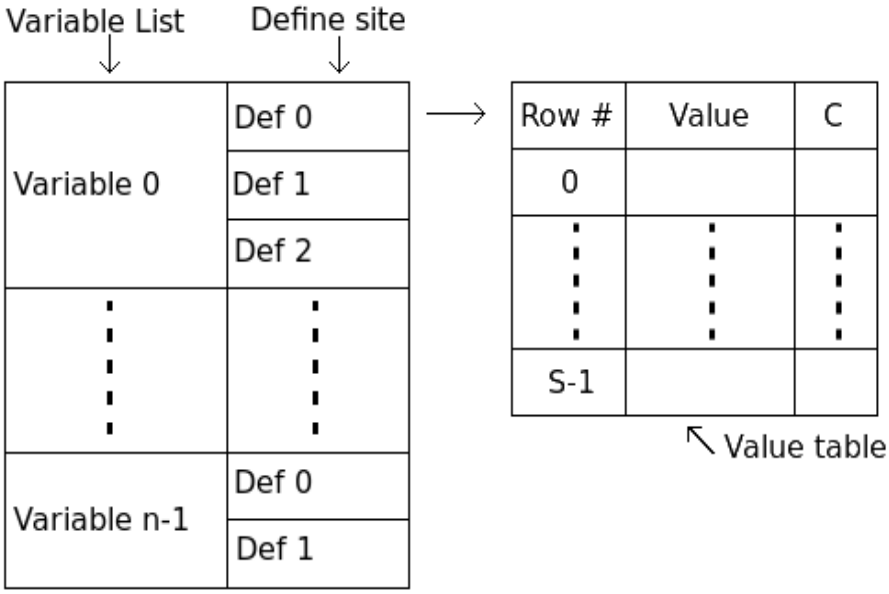

Figure 2.3: Internal data structure of code value profiling

Value profiling for code specialization

The same technique used for method specialization can be roughly used for code specialization: here you simply have to change the table and you will obtain something like what is shown in figure 2.3. Now for each variable in a method you have to register the definition sites, i.e. where the variable is used on the left side of an assignment, and this can be computed very easily. You have to associate a value table which keeps track of the values the variable has assumed more frequently to each row of this variable list table. The same replace policy used for value table of method value profiling can be used here. Te have to insert profile call in every place where a variable is defined to obtain data for this table.

However, this way of proceeding gives hints to the specializer only if a variable is constant like or non-constant like. In code specialization we are interested in finding the starting point of a trace. To do so we will follow the approach described in [4]. First of all we have to identify the number of occurrences of a trace in a

program execution, formally:

$$occurrence(t) \equiv t \text{ is followed on an execution of } f$$

We can calculate the probability of this statement:

$$\mathbf{Pr}[occurrence(t)] = \prod_{edge\ e=(m,n)\in t} \frac{count(e)}{count(m)}$$

Together, the traces and their probabilities constitute f's *expected execution set*, or *EES*, dictated by the profile. We can now define the *influence* of an instruction i with respect to a particular trace t as the length of the subtract from the first occurrence of i along t until the end of the trace:

$$influence_i(t) \equiv length(e_k...e_n)$$

where e_k is the edge from the first occurrence of i. The overall influence of i is the expected length of this path over all traces, computed as the influence per trace, weighted by each trace's probability of occurring:

$$influence(i) \equiv \sum_{t\in EES} \mathbf{Pr}[occurrence(t)] \cdot influence_t(i)$$

Unfortunately, the existence of loops in control flow graphs makes this computation of influence infeasible because the EES can be infinite in size. Instead, we can recast the influence of the instruction i as the combination of two problems: the probability of reaching i during the invocation of f (*reach*) and the expected length from i to the end of the function once i is reached(*len*):

$$\mathbf{Pr}[reach(m,s)] = \sum_{edge\ e=(m,n)} \frac{count(e)}{count(m)} \cdot \begin{cases} 1 & \textbf{if } n = s \\ \mathbf{Pr}[reach(n,s)] & \textbf{otherwise} \end{cases}$$

$$\mathbf{E}[len(m)] = 1 + \sum_{edge\ e=(m,n)} \frac{count(e)}{count(m)} \cdot \mathbf{E}[len(n)]$$

The problems are both recursive definitions that produce systems of linear equations and the following definition of influence:

$$influence(i) = \mathbf{Pr}[reach(start,i)] \cdot \mathbf{E}[len(i)]$$

We can use execution count data to simplify the problem further. We solve just one system of linear equation, by directly computing the expected number of time i is executed *per invocation* of $f(num)$. We can then view each trace as a series of shorter paths from an instance of i to immediately next instance, or (in the case of last short path) to the end of the function, and define a system of equations to compute the expected length of such a path($slen$)

$$\mathbf{E}[num(m)] = \frac{count(m)}{count(start)}$$

$$\mathbf{Pr}[slen(m,s)] = 1 + \sum_{edge\ e=(m,n)} \frac{count(e)}{count(m)} \cdot \begin{cases} 0 & \textbf{if } n = s \\ \mathbf{E}[slen(n,s)] & \textbf{otherwise} \end{cases}$$

The influence of i is the product of these two equations:

$$influence(i) = \mathbf{E}[num(i)] \cdot \mathbf{E}[slen(i,i)]$$

Once the influence has been calculated, we can select the instruction with the highest value (or with the value over a given threshold) and select this as trace start. As shown in [4], influence succeeds at identifying good dispatch instructions both accurately and consistently.

Value profiling for memory locations

Value profiling can be applied not only to variables and registers but also to memory location. Loads from memory can be safely removed if the specializer knows that the value of those memory locations will not change throughout the rest of the program execution. As an alternative, the specializer can employ the more aggressive strategy of assuming that certain locations will remain constant. This strategy uncovers more constants, those for which the guarantee of invariance is impossible or very difficult to acquire.

However, if such an optimistic assumption turns out to be false – an assumed constant memory location is actually updated later in the execution – any optimization that depends on it must be invalidated. Thus, since the cost of an

incorrect assumption is high, a technique for making accurate optimistic guesses that are most often right is needed. Ideally we would like to have an information as the following one:

$$const(a) \equiv \text{ Will address } a \text{ be updated in the remainder of execution?}$$

While a number of static analysis can conservatively approximate this predicate, the specializer that works in a dynamic compiler requires a technique as efficient as possible, and it would also be very appreciate to exploit as many constants as possible, even those that may not reveal themselves in a static analysis. So, we use the past behaviour of the program, from the start of the execution until the specialization time, as a guide to future execution. We simply assume that if a location has been constant, it will remain constant. This approximation of the memory location *const(a)* is efficient but not conservative:

$$var(a) \equiv \text{ has } a \text{ been written to more than once?}$$

If *var(a)* holds, then a is assumed to remain variant for the rest of the execution; if not, the a has only been initialized and we optimistically consider it constant. Monitoring every store is too expensive, hence, the specializer could employ sampling-based profiling which only one out of 1000 or so stores is monitored. The profiler thus evaluates the following predicate, which approximates *var(a)*:

$w(a) \equiv$ Does a random sample of observed stores contain a store to address a?

If so, than a has almost certainly not been constant, since it must been updated enough to be detected by the profiler. Furthermore, rarely written locations are likely to be identified as constants, which can allow for beneficial specializations until the next time they are updated.

Listing 2.7: C example for method costs and benefits analysis from [5]

```
int main(){
   int i, a, b, k, m, c[100], d[200], e, result = 0;
   /* .. Other variables initialization ... */
   result = foo(a, 100, c, k);
   for(i = 0; i < 10; i++){
      result += foo(i, 100, c);
      result += foo(b, e, d);
   }
}
int foo(int fa, int fb, int *fc){
   int i, j, sum = 0;
   for(i = 0; i < fb; i++)
      for(j = 0; j <  fb/2; j++)
         sum = fa * fc[i];
   return sum;
}
```

2.2.5 Costs and benefits evaluation

After the profiling (or as an alternative the binding-time analysis) phase has found some constant like parameters or variables we have to understand if specialization can actually improve performance. For this reason the specializer has to perform a cost and benefit evaluation. Here the program that performs partial evaluation can use a large number of different heuristics, each that could possibly work well in a given context. Most of these heuristics works on information obtained from profiling, so here we will discuss just this case.

Costs and benefits from method profiling

In order to execute this analysis you have to suppose that the specializer has executed the method profiling described in the previous section (See 2.2.4). First of all the specializer represents all possible common cases as hierarchical tree based on profiling information and prunes out the cases which are expected to show only marginal improvement even after specialization. Suppose, for example, you want to specialize the program described in listing 2.7 assuming that variable `b` (the

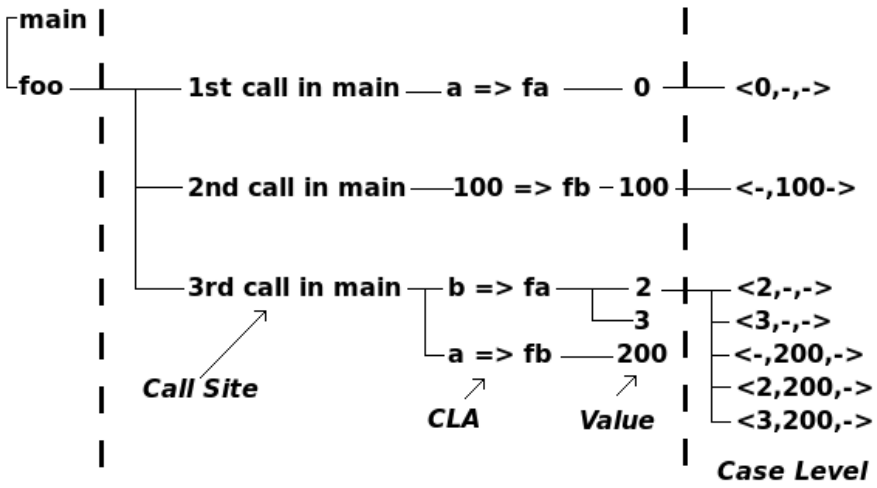

Figure 2.4: Hierarchical tree representation of commons uses

first parameter of the third `foo` call) has two common values – 2 and 3. The specializer should obtain a method tree like the one in figure 2.4. The first level of the tree represents the name of the function that must be specialize. In the second level, for each function, you find the call site level: in this example there are three call sites for method `foo` because it is called three times in function `main` while there are no call sites for `main` because it has never been called by any other method. The call site level has two-level sub-hierarchies to represent the function arguments that are constant like (in this context they are called *CLA – Common Level Argument*) and their common values. *CLA level* represents the mapping between CLA parameter and its corresponding formal parameter and *value level* is used for common values CLAs. In case level, common values are related to each formal parameter by positional mapping and "-" represents *do not care*: the parameter value in that position is not considered in this case. Note that there are (in this example) seven possible cases, even thought the number of call sites are only three. The over all size of the search space to find common cases is calculated by summing the size of search space for each call site. And the number of cases examined at each call site is one less than the product of the

number of common values for all CLAs:

$$|B_{ij}| = \prod_{k=0}^{|A_{ij}|-1} (|V_{ijk} + 1| - 1)$$

$$|S| = \sum_{i=0}^{|P|-1} \sum_{j=0}^{|C_i-1|} |B_{ij}|$$

where P is the set of all procedure, $\mathtt{C}_i$ the set of call site for procedure i, $\mathtt{A}_{ij}$ the set of all CLA for call site j of procedure i, $\mathtt{V}_{ijk}$ the set of values for CLA k in call site j for procedure i and $\mathtt{A}_{ij}$ is the set of cases for call site j of procedure i.

Once the common cases tree has been build the specializer can try to prune it in order to skip the specialization of cases that would not increase performance. The search space reduction is performed basing on *Normalized Computation Effort* (*NCE*). The computation effort of each procedure is obtained from execution frequency profiling. Based on this , NCE of each common case can be estimated in a hierarchical order. In other words, NCE of each procedure is estimated first and then NCE of each call site is calculated and so forth. NCE, in a hierarchical tree, represents the maximum degree of improvement that has to be obtained by specializing all cases belonging to the given node. For pruning purpose, the specializer user constraint, called *computational threshold* (*CT*), is defined in terms of NCE.

Table 2.1: Example of profiling results for listing 2.7

p_i			c_{ij}				a_{ijk}			
i	proc	NCE	j	site	freq.	NCE	k	var	freq	NCE
0	main	5%	0	-	1	5%	-	-	-	-
1	foo	95%	0	1st	100	8%	0	a	0	100
			1	2nd	1000	29%	1	100	100	10000
			2	3rd	1000	54%	0	b	2	1000
									3	8000
							1	e	200	10000

First of all the specializer performs *procedure level pruning*. The NCE of each procedure is obtained by normalizing its computational effort to the total

computation effort. For example, consider the profiling result for the program in listing 2.7 shown in table 2.1. Consider $CT = 0.1$: since the NCE of procedure `main` is lower than computation threshold it is removed from the hierarchical tree. The NCE of `main` is computed as follows: $NCE_{main} = \frac{1}{2101} = 0.05 = 5\%$. What is more, the procedures which do not have any descendants are eliminated. The pruning at this level has the largest impact on reducing the search space.

A similar pruning is performed at *call site level*. The NCE of each call site can be computed in the same way of procedure pruning. In table 2.1, the first call of procedure `foo` will be out because its NCE is less than CT (that we will suppose to be equals to 0.1 again). We also consider NCE for two sub-hierarchies in call-site level. NCE of each CLA is calculated by weighting the NCE of the corresponding procedure call (c_{ij}) by its observed ration (OR_i):

$$NCE(a_{ijk}) = NCE(c_{ij}) \times \sum_{k=0}^{|A_{ij}|-1} OR_k$$

In addiction, NCE of each common value can be computed similarly. For example, consider the third call of procedure `foo`, where a_{120} has two common values – 2 (v_{1200}) and 3 (v_{1201}). We can also compute the observation ration for each value: $OR(v_{1200}) = \frac{100}{10000} = 0.1$ and $OR(v_{1200}) = \frac{8000}{10000} = 0.8$. Thus, $NCE(a_{120}) = 0.54 \times (0.1+0.8) = 0.468$ which is larger than CT so a_{120} will not be pruned at CLA level. At value level, $NCE(v_{1200}) = NCE(1_{120}) \times OR(v_{1200}) = 0.428 \times 0.1 = 0.0468$ which is smaller than CT and v_{1200} is pruned out, whereas v_{1201} is not eliminated because its NCE is larger than CT.

We can now prune level at case level too. NCE of each case can be calculated using NCE of common values. Because at this level each common case can depend on different common values (for example in the case $< 2, 200, - >$) NCE should be computed in a different way, multiplying NCE of common values which are involved in forming the case:

$$NCE(b_{ijm}) = \prod_{k=0}^{|B_{ij}|-1} NCE(cv_K)$$

where $NCE(cv_k) = \begin{cases} 1 & \textbf{if } v_{ijkl} = \text{“-”} \\ NCE(v_{ijkl}) & \textbf{otherwise} \end{cases}$

For example, look at the case $b_{120} =< 2, 200, - >$ which is a child of c_{12} (third call in `main`) and c_{12} is also a child of p_1 (procedure `foo`). As we already said $NCE(v_{1200}) = 0.0468$. Similarly, $NCE(v_{1210}) = 0.54 \times \frac{10000}{10000}$. From the previous equation we obtain $NCE(b_{120}) = 0.0468 \times 0.54 = 0.027$, thus it is dropped from the search space. But this pruning would not happen in practice because v_{1200} is already pruned out at *value level*. Moreover, notice that the case $< -, 200, - >$ which has less information than $< 2, 200, - >$ (from the viewpoint of specializer) is still in the tree due to its high NCE that is 0.54.

In order to reduce the search space further, we can define *dominated cases* that can be eliminated from the search space. We say that b_{ijm} is dominated by b_{ijt} if all common values of b_{ijm} appear in b_{ijt} and $NCE(b_{ijt}) \geq NCE(b_{ijm})$. A dominated case does not need to be specialized because it has less information and is less important in terms of NCE than dominant case.

After all cuts have been performed at each level the remaining common cases can be actually used by the specializer to perform partial evaluation on the found values.

Costs and benefits for code profiling based on saved cycles

Now we are in the case where we want to create a trace. Assume that the specializer has identified a suitable dispatch point instruction i and one of its “hot values” v, as already shown in section 2.2.4. The specialization procedure creates a specialized trace starting from i. The procedure uses a simple benefit analysis to identify when to end the trace; the relevant values are the *net benefit* of a trace, the estimated total number of run time cycles it will save, and the *instantaneous benefit*, an estimate of the benefit that will accrue from growing the trace further.

Consider now a trace t of a hot value v starting at instruction i. Its *pure benefit*, *Pure(t)*, is an estimation of the number of dynamic cycles it will save,

using past execution frequencies to guess at the future:

$$Pure(t) = (C + wE) \times count(v)$$

where C is the number of inexpensive instructions, such as ALU operations, E is the number of expansive operations, like loads and bounds checks, that have been optimized away, and w is a constant weighting factor to account for the fact that these latter instructions take longer to execute; $count(v)$ is an estimate of the number of times this trace will run in the future.

The *net benefits* of a trace, *Net(t)*, is the pure benefit minus the cost of recompilation (once a trace is created, it must still be compiled down to machine code), invalidation and dispatching:

$$Net(t) = Pure(t) - R \times length(t) - I(t) - count(p)$$

where R is a constant recompilation factor, $I(t)$ is a function of the number of invariant memory location that must be monitored (you can put it to zero if you do not consider the profiling for memory explained in section 2.2.4), and $count(p)$ accounts for the cost of dispatch: an instruction must be executed every time the dispatch point is reached. The net benefit of a dispatch point is the sum of the net benefits of its traces.

When creating a specialized trace, we would like to know how well the procedure is currently doing, to help to decide when to stop specializing. Given a window of n previous instructions in the optimized trace, we can define the *instantaneous benefit*, I as:

$$I = \frac{((C_n + wE_n) \times count(v))}{n} - R$$

The instantaneous benefit is a quick estimate of the current average benefit we are receiving per instruction.

Costs and benefits for code profiling based on instruction latency

Another way to compute costs and benefits can be found in [6]. Suppose here that a program point p and a value v for a register r useful for specialization are known. For each instruction i that will be part of the trace there may be some benefits in knowing the value v. The magnitude of this benefit can depend on the type of instruction i, so we can calculate the *Savings(i,r)*:

$$Savings(i, r) = \textbf{if } Evaluable(i, r) \textbf{ then } Latency(i) \textbf{ else } 0$$

where $Evaluable(i, r)$ is a function that is `true` when you know the value of the register r and you can computer the instruction i at specialization time, $Latency(i)$ gives the latency of the given instruction type (obviously this definition is very architecture dependent).

Now we can compute the direct and indirect benefits of a given instruction as follows:

$$Benefit(p, r) = \sum_{i \in Uses(p,r)} [Freq(i) \times Savings(i, r) + IndirBenefit(i, r)]$$

$$IndirBenefit(i, r) = \begin{cases} Benefit(Next(i), ResultReg(i)) & \textbf{if } Evaluable(i, r) \\ 0 & \textbf{otherwise} \end{cases}$$

where:

- *Uses(p,r)* is the set of all instructions that use the value of register r that is available at program point p;
- *Freq(i)* refers to the dynamic execution frequency of the instruction;
- *Next(i)* is the instruction following i;
- *ResultReg(i)* is the register where instruction i puts his result.

Looking at the equation given for $Benefit$ and $IndirBenefit$ it is possible to see that the benefit is referred to a particular register r. The problem is that

Listing 2.8: Algorithm for approximate benefit computation as shown in [6]

```
for each unprocessed instruction i do
    if i is not a memory operation do
        j = \textit{defInst}(i);
        if j ≠⊥ and all instructions dependent on i have been processed do
            Benefit(j)+ = Benefit(i) + Savings(i, ResultReg(j));
            mark i as processed;
        end if
    end if
end for
```

it could be difficult to calculate at run time due to time-constraints problems. For this reason we should find a way to approximate the solution. First of all, let *defining instruction* of an instruction i, written $defInst(i)$, be the (single) instruction j such that knowing the value computed by j into its destination register allows the specializer to determine the value computed by i; if there is not such a single instruction, the defining instruction is undefined, denoted by $\perp$. Use-definition chains are used to compute the defining instruction for an instruction $i \equiv r_a \oplus r_b$ as follows:

1. if the values of both r_a and r_b are statically known, $defInst(i) = \perp$;
2. otherwise, if the value of one of the operand registers is statically known, and there is a single definition of j for the other operand register that reaches i then $defInst(i) = j$;
3. otherwise, if $r_a = r_b$ and there is a single definition j for r_a that reaches i, then $defInst(i) = j$;
4. otherwise $defInst(i) = \perp$.

The benefit for each instruction can now be computed as follows. Suppose that $Benefit(i)$, where i is an instruction, denotes the value $Benefit(p, r)$ where p is the program point immediately after i and r the destination register of i. First of

all we mark all instructions in the program as *unprocessed*, and set $Benefit(i) = 0$ for each instruction i. Now we can execute algorithm shown in listing 2.8.

The algorithm will not process any instruction that is involved in such a cycle, since $Benefit(i)$ is added to $Benefit(j)$ only after all the instructions dependent on i have been processed. This will cause benefit information to not be propagated around loops, making the cost for estimating benefits simpler and faster.

2.3 Related optimizations

As already explained, program specialization relies on many other optimization techniques. In this section how different optimizations could improve specialization performance is going to be explained. For a more exhaustive explanation of other techniques see [14, 15, 16, 17].

Constant folding and constant propagation

Constant folding and *constant propagation* are related optimization techniques used by many modern compilers.

Constant folding is the process of simplifying constant expressions at compile time. Terms in constant expressions are typically simple literals, such as the integer 2, but can also be variables whose values are never modified, or variables explicitly marked as constant. Consider the statement:

```
i = 320 * 200 * 32;
```

Most modern compilers would not actually generate two multiply instructions and a store for this statement. On the contrary, they identify constructs such as these, and substitute the computed values at compile time (in this case, 2,048,000), usually in the intermediate representation (IR) tree. In some compilers, constant folding is done early so that statements such as C's array initializers can accept simple arithmetic expressions. However, it is also common to include further constant folding rounds in later stages in the compiler. Constant folding can be

done in a compiler's front end on the IR tree that represents the high-level source language or in the back end, as an adjunct to constant propagation.

Constant propagation is the process of substituting the values of known constants in expressions at compile time. Such constants include those defined above, as well as intrinsic functions applied to constant values. Consider the following pseudo-code:

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

Applying constant propagation once yields:

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

A typical compiler might apply constant folding now, in order to simplify the resulting expressions, before attempting further propagation. Constant propagation can also cause conditional branches to simplify to one or more unconditional statements, when the conditional expression can be evaluated true or false at compile time to determine the only possible outcome.

These two techniques are indispensable in order to obtain benefits in partial evaluation because, once you know constant like value at specialization time, the specializer can run both these optimizations and spread static information all over the method.

Dead code elimination

Dead code elimination is an optimization that removes code that does not affect the program. Removing such code has two benefits:

- It shrinks program size, an important consideration in some contexts.
- It allows the running program to avoid executing irrelevant operations, which reduces its running time.

Dead code includes code that can never be executed (unreachable code), and code that only affects dead variables, i.e, variables that are irrelevant to the program. Most advanced compilers have options to activate dead code elimination, sometimes at varying levels. A lower level might only remove instructions which cannot be executed. A higher level might also not reserve space for unused variables. However a higher level might determine instructions or functions that serve no purpose and eliminate them.

In practice, much of the dead code that an optimizer finds is created by other transformations in the optimizer. For example, the classic techniques for operator strength reduction insert new computations into the code and render the older, more expensive computations dead. Subsequent dead code elimination removes those calculations and completes the effect (without complicating the strength-reduction algorithm).

Also partial evaluation can change the code making some instructions become unreachable so it's highly encouraged the use of dead code elimination after the specializer has completed his job.

Inlining expansion

Inline expansion, or *inlining*, is an optimization that replaces a function call site with the body of the callee. This optimization may improve time and space usage at run time, at the possible cost of increasing the size of the final program. Although inlining removes the cost of the function call and return instructions, since it eliminates overhead from calls, these are often small savings. The major savings often come from the additional optimizations that become possible on the inlined function body: for example, a constant passed as an argument can often be propagated to all instances of the matching parameter. Inlining may increase the size of the generated code, though it does not always do so. Inlining may make the generated code slower as well: for instance by decreasing locality of reference.

Inline expansion itself is an optimization but it is much more important as an

enabling transformation. That is, once the compiler expands a function body in the context of its call site – often with arguments that may be fixed as constants – it may be able to do a variety of transformations that were not possible before. For example, a conditional branch may turn out to be always true or always false at this particular call site. This, in turn may enable dead code elimination, loop-invariant code motion, or induction variable elimination.

Replacing a call site with an expanded function body can however worsen performance in several ways:

- In applications where code size is more important than speed, such as many embedded systems, inlining is usually disadvantageous except for very small functions;
- The increase in code size may cause a small, critical section of code to no longer fit in the cache, causing cache misses and slowdown;
- The added variables from the inlined procedure may consume additional registers, and in an area where register pressure is already high this may force spilling, which causes additional RAM accesses.
- A language specification may allow a program to make additional assumptions about arguments to procedures that it can no longer make after the procedure is inlined;
- If code size is increased too much, resource constraints such as RAM size may be exceeded, leading to programs that either cannot be run or that cause thrashing. Today, this is unlikely to be an issue with desktop or server computers except with very aggressive inlining, but it can still be an issue for embedded systems.

Compiler developers typically keep these issues in mind, and incorporate heuristics into their compilers that choose which functions to inline so as to improve

performance, rather than worsening it, in most cases. Furthermore, it is not always possible to inline a subroutine. Consider the case of a subroutine that calls itself recursively until it receives a particular piece of input data from a peripheral. Because the halting problem is undecidable, the compiler cannot generally determine when this process will end so, if it was designed to inline every single subroutine invocation it would never finish inlining. Thus, compilers for languages which support recursion must have restrictions on what they will automatically choose to inline, to avoid getting stuck in an infinite regress.

As previously mentioned in this chapter, inlining is an optimization technique that can enhance partial evaluation performance since specialization can significantly reduce the size of a method. Inline expansion can therefore suppress the cost for method call that can be grater than the whole cost of method execution.

Strength reduction

Strength reduction is an optimization technique where a costly operation is replaced with an equivalent, but less expensive operation.

Operator strength reduction involves the use of mathematical identities to replace slow math operations with faster operations. Examples of this include: replacing integer division or multiplication by a power of 2 with a shift operation, replacing integer division by a constant with reciprocal multiplication, and replacing integer multiplication by a constant with a combination of shifts, adds or subtracts. The cost and benefits will depend highly on the target CPU and sometimes on the surrounding code (depending on the availability of other functional units within the CPU).

Also this last technique can gain much more benefits if it's executed after the specialization process because it could potentially find other operations to reduce not available at compile time.

2.4 Existing specializing compilers and tools

Some compilers and tools have already been build in order to show how program specialization can enhance program execution performance. Some of them are real compilers (for example Psyco [13, 18]) while others are simple tools that automate the specialization process aiding the specializer to understand which parameters of a function are constant like (for example Calpa [8]). now a short dissertation about these compilers will be presented. It is possible to see a quick comparison of all these compilers and tools respect to the different phases of partial evaluation in table 2.2.

Table 2.2: Comparison of exisisting compilers and tools for partial evaluation respect online or offline

Tool Name	Pre-Processing	Invariant Detection	Cost and Benefits Analysis	Code Specialiaztion	Dispatch
Tempo	off-line	off-line	-	off-/on-line	off-/on-line
DyC	off-line	off-line	-	off-/on-line	off-/on-line
Calpa	off-line	off-line	-	Uses DyC	Uses DyC
Psyco	on-line	-	on-line	on-line	on-line

Tempo Tempo [10, 19] is a partial evaluator for C programs. Tempo is an off-line specializer; it is divided into two phases: analysis and specialization. The input to the analysis phase consists of a program and a description of which inputs will be known during specialization and which will be unknown. Based on this knowledge, dependency analyses propagate information about known and unknown values throughout the code and produce an annotated program, indicating how each program construct should be transformed during specialization. Because C is an imperative language including pointers, the analysis phase performs alias and side-effect analyses in addition to binding-time analyses. The accu-

racy of these analyses is targeted towards keeping track of known values across procedures, data structures and pointers. Following the analysis phase, the specialization one generates a specialized program based on the annotated program and the values of the known inputs. Tempo can specialize programs at compile time as well as run time.

DyC DyC [20] is an annotation-driven, selective dynamic compilation system that attains speedups of up to 4.6x on a group of medium-sized C programs. To achieve this level of performance, DyC contains a sophisticated form of partial-evaluation-style binding time analysis (BTA). To trigger dynamic compilation, programmers annotate their source code to identify constant like variables on which many calculations depend. DyC's BTA analyzes code downstream of the annotations, intraprocedurally, to separate those computations that depend solely on the annotated constant like variables (*static computation*), along with additional constant like variables that are derived from them, from the computations that depend at least in part on run-time variables (*dynamic computation*). Static computations are executed just once, at dynamic compilation time. For each trace found, DyC builds a custom dynamic compiler that generates code at run time, using the values of the constant like variables once they become known. To minimize dynamic compilation overhead, DyC performs much of the analysis and planning for dynamic optimization during static compile time, freeing the custom dynamic compiler from needing an intermediate representation and iterative analyses at run time. Invoking the dynamic compiler and dispatching to dynamically generated code are the principal sources of run-time overhead.

Calpa Calpa is a tool that tries to automate the annotating process in DyC compiler. Calpa combines program analysis and profile information to derive au-

tomatically the annotations that drive dynamic compilation. Calpa consists of two modules: an instrumentation tool and an annotation selection tool. The programmer first runs the instrumentation tool on the program to produce an instrumented version of the program that will generate and summarize value and frequency data. The instrumented version is then executed on some representative input, yielding an execution profile of this run-time information. The programmer then invokes the annotation selection tool on the program and the profile. The annotation selector searches the space of possible dynamic compilation annotations, using an internal model of the costs and benefits of dynamic compilation and the execution profile to estimate the overall impact of each candidate annotation. The selection tool reports its choices by producing an annotated program. This annotated program is then compiled by DyC to yield a final executable program that contains specialized dynamic compilers.

Psyco Psyco is a just-in-time representation based on specializer operating on the Python language. It is not an independent tool: it is an extension module, written in C, for the standard Python interpreter. It generates machine code by writing the corresponding bytes directly into executable memory. Psyco uses the actual run-time data that the given program manipulates to write potentially several versions of the machine code, each differently specialized for different kinds of data. There is no static analysis of the program, no separated compilation phase, no type inference. It is all done at run-time: Psyco infers, from the values the program manipulates, some restrictions about the variables, like always containing an integer, or a list of strings of length 1, or a tuple whose first item is zero. Using these restrictions, efficient machine code can be emitted. Only this second phase – code generation – is similar to what a C or a JIT compiler does. The actual performance gains can be very large. For common code, expect at least a 2x speed-up, more typically 4x. But where Psyco shines is when running algorithmic

code — these are the first pieces of code that you would consider rewriting in C for performance. You might get 10x to 100x speed-ups. Because of the nature of Psyco, it is difficult to forecast the actual performance gains for a given program. The main drawback of Psyco is that it uses too much memory.

Chapter 3

Dynamic compilation

A *dynamic compiler* [21] (also called *just in time – JIT –* compiler) has a very different design criteria if compared to a static one. Because compilation occurs at run time in a dynamic compiler, it is imperative to minimize the resource usage. A dynamic compiler should also be able to effectively utilize the extra information about how the program is actually running – for example using profile information, run time data values, applying specialization, etc. Finally, if the system support loading code dynamically, the compiler should be able to deal with incomplete knowledge of the program.

The *dynamic compiler* converts code at run time before executing it natively, for example bytecode into native machine code. The performance improvement over interpreters originates from caching the results of translating blocks of code, and not simply from reevaluating each line or operand every time it is met. It also has advantages over statically compiling the code at development time, as it can recompile the code if this is found to be advantageous, and may be able to enforce security guarantees. Thus JIT can combine some of the advantages of interpretation and static (ahead-of-time) compilation (see section 3.1.1).

Several modern run time environments, such as *Microsoft's .NET Framework* and most implementations of *Java* rely on JIT compilation for high-speed code execution.

3.1 Overview

In a bytecode-compiled system (like in Java or .NET environment), source code is translated into an intermediate representation known as bytecode. Bytecode is not the machine code for any particular computer, and may be portable among computer architectures. Moreover the bytecode may be interpreted, or run, on a virtual machine. A just in time compiler can be used as a way to speed up execution of bytecode. By the time the bytecode is run, the JIT compiler will compile some or all of it into native machine code for better performance. This can be done per-file, per-function or even on any arbitrary code fragment; the code can be compiled when it is about to be executed (hence the name "just in time").

On the contrary, a traditional interpreted virtual machine will simply interpret the bytecode, generally with much lower performance. Some interpreters even interpret source code, without the step of first compiling to bytecode, with even worse performance. Statically compiled code or native code is compiled prior to deployment.

A common goal of using JIT techniques is to reach or surpass the performance of static compilation, while maintaining the advantages of bytecode interpretation. Much of the "heavy lifting" of parsing the original source code and performing basic optimization is often handled at compile time, prior to deployment: compilation from bytecode to machine code is much faster than compiling from source. The deployed bytecode is portable, unlike native code. Since the run time has control over the compilation, like interpreted bytecode, it can run in a secure sandbox. Compilers from bytecode to machine code are easier to write, because the portable bytecode compiler has already done much of the work.

JIT code generally offers far better performance than interpreters. In addition, it can in some or many cases offer better performance than static compilation, as many optimizations are only feasible at run-time, for example:

1. The compilation can be optimized to the targeted CPU and the operating system model where the application runs. For example JIT can choose MMX CPU instructions when it detects that the CPU supports them. With a static compiler one must write two versions of the code, possibly using inline assembly;

2. The system is able to collect statistics about how the program is actually running in the environment it is in, and it can rearrange and recompile for optimum performance. However, some static compilers can also take profile information as input;

3. The system can do global code optimizations (e.g. in-lining of library functions) without losing the advantages of dynamic linking and without the overheads inherent to static compilers and linkers. Specifically, when doing global inline substitutions, a static compiler must insert run-time checks and ensure that a virtual call would occur if the actual class of the object overrides the inlined method;

4. Although this is possible with statically compiled garbage collected languages, a bytecode system can more easily rearrange memory for better cache utilization.

However, JIT typically causes a *slight delay* in the initial execution of an application, due to the time taken to compile the bytecode. Sometimes this delay is called "start-up time delay". In general, the more optimization JIT performs, the better code it will generate. However, users will experience a longer delay. A JIT compiler, therefore, has to make a trade-off between the compilation time and the quality of the code it hopes to generate.

3.1.1 Dynamic translation and optimization

The field of dynamic compilation covers a range of topics linked by the common theme of generation of executable code for a program at run time. We can divide this general field into two areas, defined by the goal achieved through dynamic compilation: first, dynamic compilation can be used when the code requires information only available at run time to be compiled; second, it can be used to exploit some degree of optimization not available at compile time. The first application usually falls under the name of *Dynamic Translation*, while the second goal is simply known as *Dynamic Optimization*. Of course, both goals may be aimed at a single dynamic compiler, since, usually, a dynamic translator will be supported by optimization phases.

The goal of Dynamic Translation is easily defined: in presence of a program written in some form, a dynamic compiler must generate the equivalent program for its target architecture. At a first look, a static compiler may seem the best choice for this task, as there is no interest in obtaining run time information for optimization - the primary goal is compatibility. What is more, this type of code often does not modify itself, so that once a translation is established it can be safely reused. What makes necessary the use of dynamic compilation is that, in fact, the translation often cannot be performed at compile time. Dynamic compilation is also used to translate legacy code written in the executable code of a different hardware machine, in a portable code, written in an abstract bytecode format. In the past, it was also used for code compaction purposes, as an higher level code was decomposed at run time in a much larger target code, which was then discarded after execution. In this way that large application could be performed even on machines with small memory systems, since at any time only the space for the compact code with a fraction of the target code in addiction was present in memory.

Dynamic Optimization requires that the dynamic compiler perform some trans-

formation on the code to optimize its performance. While a dynamic compiler brings along a significant overhead, there are several ways to exploit run time information in order to compensate for this overhead and obtain increased performance. One typical optimization is based on run time constants: when a certain variable is known to assume a single value at run time, but this value is not known at compile time, it is called a run time constant. Run time constants can be subject to the same range of optimizations as compile time constants, as soon as their value is known. Therefore a dynamic optimizer would apply data-flow optimization such as constant propagation and folding, as well as more aggressive optimizations such as the specialization of functions. A typical dynamic optimizer takes control of the application, monitoring its performance and applying optimization passes when needed.

AOT and DLA compilers

Sometimes it could be very useful to generate directly executable code of some frequently used method from bytecode instead of using a just in time compiler. This way to proceed, called *Ahead of Time* (*AOT*), increase the program execution speed up because at run time you can skip the compilation and optimization phases. Obviously this could lead to portability issues and you have to recompile intermediate bytecode into machine-code for each architecture you are going to use. However, the performance of an AOT compiler is still lower than that of native code produced by a static compiler. The loss of performance can be found in two reasons: the compilation overhead and the poor quality of generated code since the compiling time minimization prevents the aggressive and costly optimizations usually performed by static compilers. These costs can be partially removed if the compiler is working in a multiprocessor environment since it can use a processor to perform compilation and optimization process while using another processor can actually execute the program code. In this way, in the best possible scenario, each invoked method has already been compiled and optimized.

To gain this performance the compiler can apply a technique called *Dynamic Look Ahead* [22]. This process works as follow: while a processor is executing a method, compilations threads (running on another processor) look ahead into the call graph, detecting methods that have good chances to be executed in the next future. Moreover, they guess whether a method is an "hot spot" in order to apply aggressive optimization or not.

3.1.2 Current trends and applications

There are several fields of application for dynamic compilation. The current trend seems to be highlighting the importance of dynamic compilation along the lines of dynamic translation for portability and mobility of the code, and for self-modifying code in simulated environments. Dynamic optimization can be subsequently employed in both cases to speed up the operation. Before dynamic optimization can be widely employed, there are still several issues to be dealt with, especially for specific application fields such as real-time application, where the assessment of the trade-offs must be accurate, and guarantees on execution time must be provided. Another major field of application for dynamic compilers is compiled simulation. While a compiled simulator usually works as a static compiler that translates object code for a simulated architecture to a host platform, inserting some instrumentation code, dynamic compilation may be used to cope with self-modifying code. Self-modifying code can be handled since the dynamic compiler can refer to the original code and translate it more than one time, if it is modified.

3.2 The .NET Framework

In this section I am going to explain briefly the *Microsoft .NET Framework* [23]. This is because in this thesis the compiler I have chosen to modify in order to test partial evaluation works on the .NET bytecode (called CIL), so I thought it would be rather interesting to understand how this framework works.

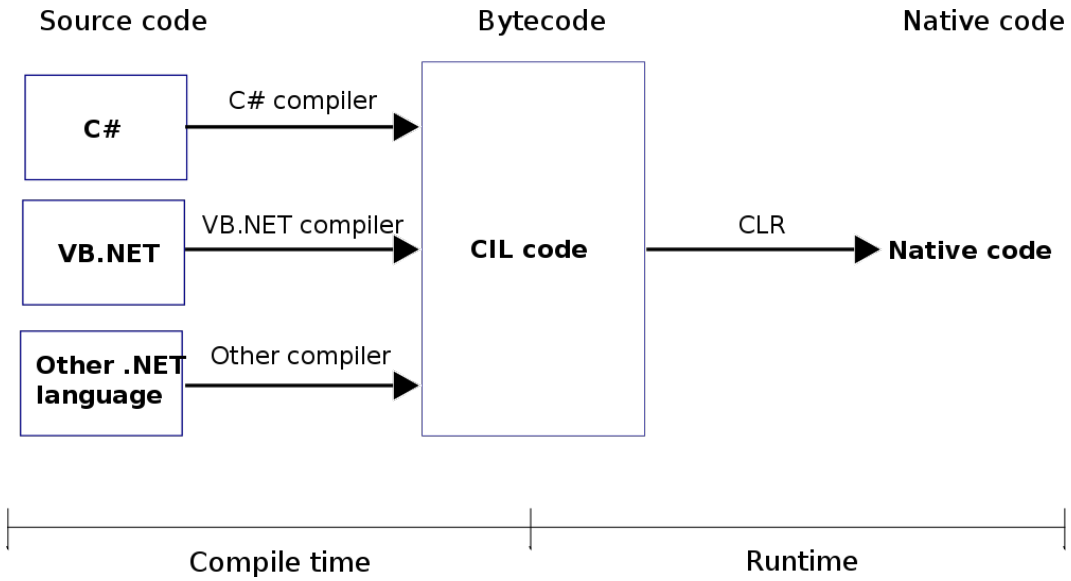

Figure 3.1: The structure of the ECMA-335 Framework

First of all, accordingly to Barry [24] the .NET framework consists of three parts:

1. the *Common Language Runtime* (*CLR*) (see section 3.2.1);

2. a comprehensive set of class library (the so called *Common Type System*);

3. class libraries associated with particular kinds of applications: console applications, windows forms applications, web form applications and web services.

This framework is actually an implementation of an open infrastructure called *Common Language Infrastructure* (*CIL*) published under ECMA-335 developed by Microsoft. It describes the executable code and Runtime environment that form the core of a number of run-times including the Microsoft .NET Framework, *Mono*, and *Portable.NET*. In the next part of this section I am going to briefly explain this infrastructure.

3.2.1 The Common Language Runtime

The *Common Language Runtime* (*CLR*) is the virtual machine component of Microsoft's .NET initiative. It is Microsoft's implementation of the *Common Language Infrastructure* (*CLI*) standard, which defines an execution environment for program code. The CLR runs a form of bytecode called the *Common Intermediate Language* (*CIL*, previously known as MSIL – Microsoft Intermediate Language). Developers using the CLR write code in a language such as C# or VB.Net. At compile time, a compiler converts such code into CIL code. At Runtime, the CLR's just-in-time compiler converts the CIL code into native code. Alternatively, the CIL code can be compiled to native code in a separate step prior to Runtime (it just compiles ahead of time as previously mentioned). This speeds up all later runs of the software as the CIL-to-native compilation is no longer necessary. The virtual machine aspect of the CLR allows programmers to ignore many details of the specific CPU that will execute the program. The CLR also provides other important services, including the following: memory management, thread management, exception handling, garbage collection, security.

In addition to Microsoft CLR implementation in the .NET framework other implementations have been built in order to work on different operating systems.

3.2.2 The Common Intermediate Language

The *Common Intermediate Language* is the lowest-level human-readable programming language in the CLI and in the .NET Framework. Languages which target the .NET Framework compile to CIL, which is assembled into bytecode. CIL is an object-oriented assembly language, and it is entirely *stack-based*. It is executed on a virtual machine.

CIL is a CPU- and platform-independent instruction set that can be executed in any environment supporting the .NET framework, and the code is verified at run time for safety, providing better security and reliability than natively compiled

binaries.

This bytecode has instructions for the following groups of tasks: load and store, arithmetic, type conversion, object creation and manipulation, operand stack management (push/pop), control transfer (branching), method invocation and return, throwing exceptions and monitor-based concurrency.

Differences between CIL and Java Bytecode

Although both Java bytecode and CIL are low-level bytecode languages that can be compiled or interpreted at runtime from a virtual machine, there are also some important differences between them. The first main difference between them is that, while CIL has been designed as bytecode language for many different languages[1] the Java bytecode has been designed specifically for the Java language so it would be rather difficult port other languages to this bytecode. Actually this isn't the complete truth. Recently some new languages have been built to reduce this gap between the .NET and JVM platforms. For example you can use two brand new languages: *Scala* [25] and *Groovy* [26]. Both can generate Java bytecode that can be run on the classic Java Virtual Machine.

Another little difference between these two languages is that CIL bytecode has more explicit delineation between value and reference types, in fact in the .NET languages there is the concept of pointer while it is lost in Java.

Other small and less important differences between these two bytecode languages are:

- NET allows user defined types that reside on the stack (`struct`) while in Java on the stack can resides only simple (primitive) types;

- .NET supports unsigned types, this makes the instruction set a bit richer;

[1] At present many languages are supported by the .NET architecture; some of these languages include C, C++, C#, J#, VB.NET, Python, Ruby and F#.

- Java includes the exception specification of methods in the bytecode. Although exception specification is usually only enforced by the compiler, it may be enforced by the JVM if a class loader other than the default one is used;

- .NET generics are expressed in CIL while Java generics only use type erasure.

3.3 The ILDJIT compiler

ILDJIT [27, 28] is a just in time distributed compiler and optimizer for the CIL described above, developed inside the "Politecnico di Milano" and released under GPL through sourceforge. Unlike many other JIT compilers, ILDJIT is designed to gain optimal execution time in a multiprocessor environment. In fact its compiling and optimizing phases are organized as a pipeline so that several methods can be simultaneously compiled by different compilation phases and each phase can have many threads that could ideally work on different processors that could even be placed on different machine reachable through a TCP/IP channel.

This compiler is very flexible because it has a plugin structure so that different modules can be reimplemented or substituted in order to select the better combination for a specific purpose. For example, you can chose a different module for optimization when you are working on an embedded mobile system. For this reason it is quite obvious that it could be easily expanded in order to support many other features like partial evaluation.

3.3.1 Execution model

The primary task of this JIT compiler is to translate each piece of CIL bytecode to a semantically equivalent target code in order to be directly executed by the hardware. For this purpose ILDJIT adopts an intermediate representation called IR. ILDJIT uses the method as *translation unit* – i.e. the smallest piece of code that can be directly translated. First of all, once a method has been loaded from

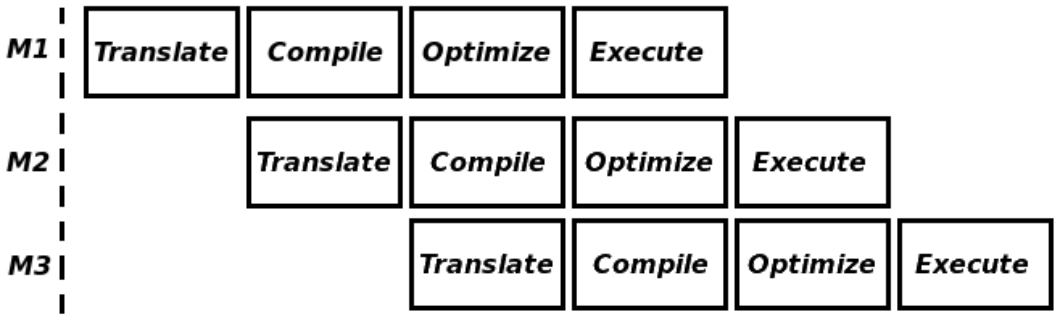

Figure 3.2: Example of the ILDJIT pipeline on three methods

a file in CIL representation, it must be converted in the IR language; IR (*Intermediate Representation*) is a low level, simple but complete language that provides a compact representation of individual methods by removing the dependencies on CIL meta-data streams. Moreover it is a machine-neutral, register-based language, such that each construct has a clear and simple meaning. For these reasons execute optimizations on a IR method is quite simple.

Since ILDJIT is just a compiler and not a complete .NET framework it does not provide itself a set of standard class libraries. Instead it uses libraries of the Portable.NET project.

Once the method has been translated, it is compiled, optimized and later executed. All these phases are managed by an internal pipeline (see figure 3.2), designed to exploit hardware parallelism at various levels: given sufficient hardware resources (i.e. given enough processors), ILDJIT can produce optimized method in machine code before the execution of the CIL program that asks them. In this case, the program execution does not need to be paused to switch to compilation.

Since the entire program is not translated into target code at once, ILDJIT has the following problem: how to handle invocations of methods not translated yet. ILDJIT redirects invocations to the internal execution engine which then calls the right internal modules responsible for translation; after that it can relink the freshly translated method to the rest of the program. This redirection mechanism is known as *trampoline* [29, 30] that has to be transparent to both

caller and callee. ILDJIT resolves the linking problem by a lazy compilation.

ILDJIT can prefetch CIL methods to translate and optimize them before their execution is demanded by the application. A critical decision is which method to compile ahead of time. ILDJIT uses a simple criterion based on *method distance* that is defined as the shortest path over the call graph between a method M_i and the current method. ILDJIT monitors the executing CIL method and computes the distance to other methods composing the application. All methods withing a threshold D are candidate for translation and optimization. The reason of the functioning of the distance method is rather obvious: a method near zero distance will be probably required in the next future execution, therefore it should be promptly translated into machine code to avoid a trampoline stall. The threshold D is adjusted at Runtime depending on available free processors.

ILDJIT allows optimizations both at IR and target code level since different algorithms for code optimization use particular features of the underlying hardware. The distinction is based on the observation that if all the semantic information of the source program is available, higher-level transformation are easier to apply. The decisions about when, where and how much to optimize are taken according to the method distance concept. This optimization phase is performed by an *optimizer* module that uses an optimization policy to reach the best level of optimization of each method undergoing translations. The policy adopted by ILDJIT works in the following way: if the execution of the application code is waiting the translation of a method (inside a trampoline), the method is placed inside the first optimisation level (that optimize very little, otherwise there is a heuristic in order to choose the right level, which is based on a back-propagation neural network trained off-line using several benchmarks. Each IR optimization available in ILDJIT is implemented as an external plugin loaded dynamically. At present ILDJIT implements different kinds of optimizations: constant propagation, copy propagation, branch prediction, dead code elimination that could be potentially

useful for partial evaluation process.

3.4 Other JIT compilers for Common Intermediate Language

ILDJIT is not the only open source implementation of the ECMA-335 standard. Many other important projects exists, in particular the most advanced are the Mono project and the Portable.NET project. In this section I am going to briefly show both these frameworks.

Mono Mono [31] is a software platform designed to allow developers to easily create cross platform applications. It is an open source implementation of Microsoft's .Net Framework based on the ECMA standards for C# and the Common Language Runtime. There are several components that make up Mono. First of all a C# Compiler that is feature complete for compiling C# 1.0 and 2.0 (ECMA), and that also contains many of the C# 3.0 features. Mono provides also a "Mono Runtime" (for CLI that gives a Just-in-Time compiler, an Ahead-of-Time compiler, a library loader, the garbage collector, a threading system and interoperability functionality), a Base Class Library compatible with Microsoft's .Net Framework and a *Mono Class Library* that add useful functionality, especially in building Linux applications. For example, with these classes, you can have access to other libraries such as Gtk+, LDAP, OpenGL, Cairo, POSIX, etc. Mono can work on many platforms (x86, x84-64, SPARC, ARM, Alpha, IA64, PowerPC) and operating systems (Windows, Linux, Unix, BSD, BSD, Mac OS X).

Portable.NET DotGNU Portable.NET [32] is an implementation of the CLI that includes everything that you need to compile and run C# and C applications which use the base class libraries, XML, and Systems.Windows.Forms. Cur-

rently supported CPUs are x86, ppc, arm, parisc, s390, ia64, alpha, mips, sparc while supported operating systems are GNU/Linux (on PCs, Sparc, iPAQ, Sharp Zaurus, PlayStation 2, Xbox,...), *BSD, Cygwin/Mingw32, Mac OS X, Solaris, AI. Because interpreting CIL bytecode directly is quite inefficient, Portable.NET takes a different approach. First, it converts the CIL bytecode into a simpler instruction set called Converted Virtual Machine (CVM). The CVM is a quite simple low-level language useful for fast compiling. The CVM approach gives it many of the benefits of a JIT compiler, in which the opcodes can be tailored to handle system differences (e.g. 32-bit vs 64-bit CPU's). At the same time, the engine's source code is highly portable to new platforms. As another optimization on some CPU's (currently x86 and ARM), this compiler further translates CVM opcodes into native machine code for direct execution. This gives an additional performance improvement with only a small coding effort required.

Chapter 4

Method specialization in a JIT compiler

In this chapter a technique to implement partial evaluation inside a JIT compiler is going to be presented. This technique is based on four main phases: *profiling*, *constant like expression detection*, *method specialization* and *dispatcher injection*.

4.1 Method and type specialization technique

To develop this technique I have followed the specialization process described in section 2.1.

The first thing that this technique requires is to understand which methods could be specialized. This *preprocessing* phase can be executed both off-line and on-line and will collect information about which method are worth specializing.

Once the specializer knows which methods could be specialized, it has to understand at run-time if it is actually useful to perform specialization. One thing that must always be kept in mind is that even if a method could be theoretically specialized, the run-time context could show that the costs of specialization are higher than the gained benefits. Another thing to remember is that it could be very useful to specialize a method in a call site of a caller method, but it could not be in another call site of another caller method. For this reason, in the second phase, the specializer will *profile* each call site where a specializable method re-

sides. In the profile phase two different profilers will be used: the first, quite small and light, will be used to understand if this call site has been executed a sufficient number of times, while the second has the objective of collecting information about actual parameters of the method called at this call site. This distinction has been made in order to try to reduce at minimum the impact of second profiler on the program time execution: before to inject the second profiler we want to be quite sure that the control flow reaches the call site a sufficient number of times.

Another profiler will be used when the specializer must profile a virtual call[1] in order to understand which actual methods will be called, depending on the dynamic type assumed by the object that calls the method. This profiler is called "zero profiler" because it should work before the first profiler.

At the end of profiling passage, if this call site will be reached many times and with parameters useful for specialization so that the call site can be *specialized*, it will be replaced by a *dispatcher*.

Sometimes the behaviour of a program can radically change. For this reason, at a given call site, where a dispatcher has been injected, if in most cases the un-specialized version of the method will be selected, then the dispatcher will decrease the performance of the program, because of a lot of un-useful comparing instructions will be executed (See section 4.4.3 for more details). For this reason it could be useful to have an *un-dispatching* phase that will remove the dispatcher and will check if other specializations are available for this call site.

Therefore, a call site can assume one of the following states in his life cycle. To better explain which states a call site can assume, an example will be shown. Consider the following simple listing:

```
void main(){
    int a = 10;
    for(int i=0;i<100000;i++){
        if(i > 50000) a = 11;
        caller(i, a);
} }
```

[1] We have to deal with virtual calls because the CIL is an object oriented language.

```
void caller(int a, int b){
   for(int j=0;i<1000;j++)
      callee(b, a+b); ✠
}

void callee(int x, int y){
   /* Do something with x and y */
}
```

First of all suppose that only method `callee` can be specialized. Now consider the call site marked by the symbol ✠. This is a non-virtual call to a static method `callee`. While the dynamic compilation process for method `caller` take place, the specializer find the call marked as ✠. Since it has no information about this call site, but only knows that it could be worthwhile specializing `callee`, the specializer assumes that the call site is *clean* (i.e. the call site is in the *clean* state).

Since the call site is in the clean state, the specializer knows that it now has to inject the first profiler in order to detect if the callee method will be called a sufficient number of times in order to justify the use of heavy second profiler and the generation of specialized method that could use a lot of memory to store the specialized code. The marked call site will be changed in something like these:

```
firstProfiler();
callee(b, a+b);
```

Once this operation has been performed the call site state will become *First Profiler Injected*.

After a while, the first-profiled call site has been possibly reached a sufficient number of times. So the `caller` method must be recompiled in order to remove the first profiler instructions and inject the second second profiler. The call site has now reached a new state: *Second Profiler Injection Request*. While a call site is in this state the first profiler has not been removed yet.

Eventually the `caller` method will be recompiled. During this compilation process the specializer will found again the `callee` call site inside the `caller`

method. Now it knows that the call site is in the *Second Profiler Injection Request* so it will remove the first profiler and will inject the second profiler code. The marked call site will reach a new state called *Second Profiler Injected* and the code will be changed in something like this:

```
secondProfiler(b,a+b);
callee(b, a+b);
```

As already mentioned the aim of the second profiler is to collect the values of the actual parameters passed to the `callee` method in order to understand which are constant like and which are non-constant like at specialization time. For this reason the function that will implements the second profiler takes exactly the same parameters as the `callee` method.

Again, after a while, the second will have collected a sufficient number of values in order to detect if a given parameter is constant like or non-constant like. For example, suppose that it find that only the first parameter of the `callee` method is constant like. For this reason the specializer changes the state of the `callee` call site to *Dispatcher Injection Request*. Moreover the specializer requires the recompilation of the `caller` method.

Eventually the caller method will be recompiled again. During this phase the specializer found again the ✠-marked call sites. Since its current state is *Dispatcher Injection Request*, the specializer knows that it has to remove the second profiler, inject the dispatcher and set the state of the call site to *Dispatcher Injected*. The resulting code generated at this specialization stage should look something like this:

```
if(b == b^constantlikevalue)
    callee_specialized(b+a) ♮
else
    callee(b, a+b);
```

When the specializer inject a new dispatcher it generates new call sites, for example here we have generated a call site that has been marked with the ♮ symbol. Anyway, since this call site has been generated by the specializer it should not be

useful specialize this call site either, for this reason the ✝-marked call site will be placed in the *Unspecializable* state.

After some times the behaviour of the program change radically. This happens when the variable `i` used inside the `for` loop of the `main` procedure exceeds the value 50000. For this reasons, once the profiler detect the change of behaviour puts the ✠-marked call site in the *Dispatcher Removal Request* state and require method `caller` to be recompiled.

Eventually the caller method will be recompiled and, when the specializer found the ✠-marked call site, since it is in the *Dispatcher Removal Request* state it will remove the dispatcher. What will happen next depends on the chosen policy of change-state (see section 4.1.1), for example we can set the call site in an Unspecializable state – which means it will never be specialized again – or the specializer may decide to restart the specialization process from one of the previous state of the call site.

What has been shown since here only work for non virtual call instruction. Now, an example that describe how to deal with virtual calls will be shown. First of all consider the following simple listing:

```
class Vehicle{
   void start(int x, int y){ /* Some code */};
}

class Car extends Vehicle{
   void start(int x, int y){ /* Some code */};
}

class Main{

   Vehicle vehicle;
   Car car;

   void main(){
      for(int i=0;i<1000;i++)
         caller(i);
   }
```

```
    void caller(int i){
      Vehicle x = vehicle;
      for(int j=0;i<1000;j++){
          if(i+j % 2 == 0) x = car;
          x.start(i,j); ◊
      }
    }
}
```

Consider now that the only specializable call site is the one marked with the ◊ symbol. In this call site the specializer to manage a virtual call, since due to polymorphism, the compiler can know only a run-time the dynamic type ob the object `x` on which the method `start` will be called.

Once, during the first compilation phase of the `caller` method, the specializer found the ◊-marked call site, since it has no information about its status it assumes that it is in the *Clean* state. For this reason if has to inject the zero profiler at this call site that will collect the dynamic types assumed by the object `x` and will set the state of the call site to *Zero Profiler Injected*. The resulting code generated at this specialization stage should look something like this:

```
zeroProfiler(typeof(x))
x.start(i,j);
```

After some times, the zero profiler will have collected a sufficient number of dynamic values assumed by the object `x` and it can detect the "constant like" dynamic types for the object `x`. The call site will then assume the *Type Dispatcher Request* state and the specializer require the `caller` method to be recompiled.

Eventually the caller method will be recompiled. When the specializer find the ◊-marked call site, it now knows it is in the *Type Dispatcher Request* state, so it will remove the zero profiler from the code, it will inject the type dispatcher and will set the state of the call site to *Type Dispatcher Injected*. The resulting code at the specialized call site should look something like this:

```
if(typeof(x) == Vehicle)
    ((Vehicle)x).start(i,j); ★
else if(typeof(x) == Car)
```

```
    ((Car)x).start(i,j); ★
else
    x.start(i,j);
```

Now the specializer has generated two other brand new call sites (both marked with the ★ symbols). Since are both non-virtual call they can be specialized as previously described.

More details about each specialization stage that will be executed will be described later in this chapter.

4.1.1 State Change Policy

Once that all the possible states for a call site has been defined, a policy that enables a call site to change his status must be selected. Though some changes of state are quite trivial – for example from the state *Dispatcher Injection Request* to the state *Dispatcher Injected* –, others are not. For example, if a method is in *Dispatcher Removal Request* state, next state can be one of: *Clean*, if we want restart from the first profiler, *Second Profiler Injection Request* if we want to restart from the second profiler or *Unspecializable Call Site* if we want to ignore this call site later in the partial evaluation process.

Obviously, all these policies are correct, but, to enable a call site to be much more reactive to the changes of the possible values of the input parameters, we have chosen that, once the method reaches the state *Dispatcher Removal Request*, next state will be *Clean*.

The chosen policy is described in figure 4.1.

Not all the change of state has already been described. Consider the *First Profiler Injected*: this state has two outgoing arrows. The first one say that the next state is *Second Profiler Request*, and this change has already been explained. On the contrary the second arrows say that a call site can directly reach the *Dispatcher Request* state. This quite un-obvious state change has been created since, sometimes, it could happen that a all actual parameters of a method are

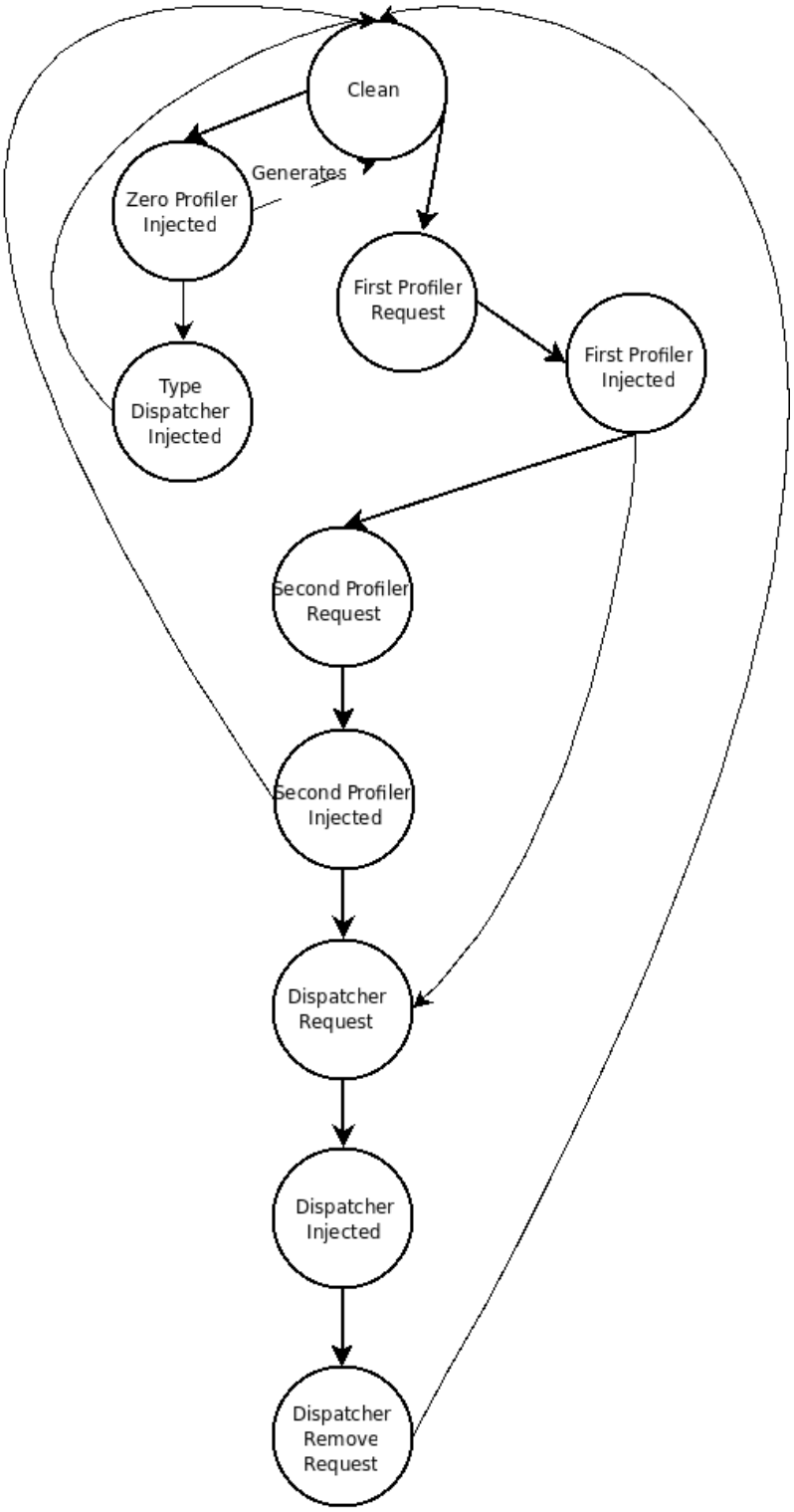

Figure 4.1: Partial evaluation state diagram

constant values, for example when there is a call of this kind: `foo(10)`. For this reason inject a second profiler would be unuseful since the specializer already know that the parameter of the method is constant like and will always assume the value 10. Here will be executed a sort of Constant Propagation through method parameters.

Anyway, following this reasoning, it would be even better directly generate the specialized method from the clean call site. This will not be done because the specializer do not actually knows if the method to specialize with only constant parameters will be ever called, and, if it directly specializes the method there will be a waste of memory, that could be better used. This fact becomes even more important when the JIT compiler is working in a embedded environment, where it should use only the strictly necessary memory.

4.1.2 Performed Specializations

For all that has been said since here, in this approach the specializer will perform three different kinds of partial evaluation:

- **Data Specialization**: the specializer tries to understand which parameters are constant like and which are not, so it can specialize a method replacing those parameters with constant values;

- **Control Specialization**: the specializer once has performed the data specialization in some cases can move a condition on parameters or generic variables outside the callee method;

- **Virtual Call Specialization**: the specializer tries to understand in a virtual call which are the "constant like" dynamic type of the objects that call the methods, and consequently which are the methods actually called [2, 33].

4.2 Preprocessing

This phase is the first the specializer should take. The target of this phase is to understand which methods could be specialized and how, without knowing anything about which parameters the method could have at run-time, but only using information available at compile time.

A simple example to better explain what will this phase produce as output is going to be shown. Suppose now you have the methods described in listing 4.1.

First of all consider `methodOne`. This method does not accept any parameters even if it could possibly do a lot of heavy stuff. Suppose it does not use any global or class static variable too. Obviously this method cannot be specialized so the preprocessing phase will not produce any output for this function.

The method `methodTwo`, on the contrary, accepts a formal parameter called `a`. As shown in the example, the method performs a very heavy mathematical computation based on the input parameter. As a consequence, if at run-time we know that `a` is constant like, we can compute the result just once and use it for all others computations reducing the time consumption. This effect can be obtained simply by using constant folding and constant propagation. So this phase should produce in output something like this:

methodTwo: #1 STATIC;

The previous line means that, for the method `methodTwo`, we should detect if the first parameter (`a`) is `STATIC`[2]. If, for a given call site, this happens, the specializer could specialize this call site considering `a` constant like. In the future, we will refer to the parameter index position simply at *position*, while the keyword `STATIC` will be called *action*.

In method `methodThree` we can see a very expansive `for` loop, which contains a switch on the parameter `a`. If we could know at run-time that `a` is constant

[2]As already mentioned static can be a synonymous for constant like, static in this case is just a short cut tag for constant like.

Listing 4.1: Examples of specializable and unspecializable methods

```
void methodOne(){
   /* ... Do some stuff ... */
}

int methodTwo(int a){
   int j, k, w, f;
   /* Do some very heavy task with variables that
      involves a */
   j = a^127;
   k = j / (a^127-1);
   w = j*a*(1/k);
   f = 1/w * 1/a;
   return a*f;
}

void methodThree(int a){
   int i;
   for(i = 0; i < HIGH_NUMBER; i++)
      if(a < 10){
         /* ... Do some stuff ... */
      } else if(a == 10){
         /* ... Do some stuff ... */
      } else {
         /* ... Do some stuff ... */
      }
}

void methodFour(int vector[LBOUND..UBOUND]){
   for(i = 0; i < HIGH_NUMBER; i++){
       check(i ≥ LBOUND);
       check(i ≤ UBOUND);
       /* Do some stuff with vector[i] */
   }
}
```

like, we could consider `a` as loop invariant, and put the switch control, not only outside the loop but also outside the method. For this reason as output we could find something like this:

```
methodThee: #1 LESS THEN 10;
methodThee: #1 EQUALS 10;
methodThee: #1 GREAT THEN 10;
```

These lines mean that we could specialize on controls, i.e. that we could put the switch control outside the `for` loop for the different expression on `a`. We have introduced three new actions `LESS THEN`, `EQUALS` and `GREAT THEN`. All these three actions have a comparison parameter associated.

In method `methodFour` we can see that the input is not a simple variable but it is an array. What we can specialize is the length, because we suppose that the content of an array could change at every call of the method; for the same reason, it is quite useless to try to specialize over pointer or data structure due to their dynamic nature. In this case, therefore, what the second profiler must collect is the size of the array parameter named `vector`. If the specializer understand that the length of the vector is constant like at run-time, we could specialize on it. In this way we could remove some bound checks, for example if we know that variable `i` of the array is always in the range `[LBOUND..UBOUND]`, it can remove the checks. The preprocessing phase should produce in output something like this:

```
methodFour: #1 LENGTH STATIC;
```

The previous statement means that the profiling phase should check weather the length of the vector is constant like or non constant like at run-time. To describe this expression another new action has been introduced: `LENGTH STATIC`.

All the presented expressions shown since here are *single expressions*, in the sense that they involve only one parameter a time. More single expressions on different parameters can be linked with the AND boolean operator. For example, for a generic method that accepts three parameters, with the first and the third specializable, we can write that:

$$\text{genericMethod: } \overbrace{\overbrace{\text{\#1 LENGTH STATIC}}^{Sub\ Expression}\ \text{AND}\ \overbrace{\text{\#3 EQUALS 10}}^{Sub\ Expression}}^{Expression};$$

A generic expression can therefore be thought as the conjunction of more *sub-expression* involving different parameters.

All these information can be extracted by a static compiler using some advanced techniques such as *symbolic program analysis* [34, 35] and *code coverage analysis* [36, 37]. As it has been described in this section this phase can be executed only off-line since it performs tasks that are very time consuming and that, therefore, will lead to a huge loss of performance of the dynamic compiler.

In order to execute this phase on-line, it must be reviewed to be much simpler. The most trivial way to work on-line, is to enable the profiling on each method. Every time the specializer finds a method without any specialization information it generate the correct specialization expressions for that method, if possible. In this approach a constant like expression can be generated for each parameter that satisfies one of the following conditions:

- The type of the parameter is a scalar value and it is not a pointer or an object;
- The parameter passed is an array of any type; also array of pointers and objects are allowed to generate an expression;

4.3 Three-level profiling

Once the specializer has gained information about which method could be specialized and how, we have now to detect if specialization is actually useful. To answer this question we have to use profiling information.

To have better results the profiler has been split in three different profilers as explained above: the *zero profiler* tries, for specializable virtual calls, to understand which method will be actually called, the *first profiler* tries to understand if

the call site is reached a significant number of times and, finally, the second profiler collect information about values assumed by actual parameters passed from the caller method to the callee method.

Every time a profiler reaches his own threshold, terminating his job, we can remove it and inject the next profiler if required (as in the case of the zero profiler and the first profiler) or go to the dispatcher injection phase (as described in 4.4).

4.3.1 Zero Profiler

The purpose of the zero profiler is to understand if a virtual call can be replaced by the dispatcher of standard call that will eliminate the virtual call overhead itself. For example, have a look at the code shown in listing 4.2: we have 4 classes (`A`, `B`, `C` and `D`), `B` and `C` override the method `A.m(int)`, and class `D` overrides method `B.m(int)`. In the body of the method `foo` there is a call to method `A.m(int)`: this is a virtual call because we actually do not know if we are calling the method `A.m(int)` or one of the overriding ones, because we know only at run-time the dynamic type of the variable `a` that could be either `A`, `B`, `C` or `D`.

To perform this operation, the specializer has to inject some code in order to collect information about the actual type of the object that calls the virtual method. It can simply store in memory the type found every time the control flow reaches this virtual call. This can be simply done by allocating a fixed amount of memory that can be treated like a circular buffer where we can place the dynamic type of the object that calls the method.

Once the profiler has collected a sufficient number of dynamic types for the object that calls the virtual method, it can inject a non-virtual call for methods which type not only has a frequency over the minimum required threshold Th_t (*type threshold*) but also that have specialization hints. We can say that the specializer injects a *type dispatcher*.

For this reason, suppose that for the code shown in listing 4.2 we see that the dynamic types (and their respective frequency) that the object which calls the

Listing 4.2: Example of virtual code before applying the zero profiler

```
class A{
   void m(int a){ ... }
}

class B extends A{
   void m(int a){ ... }
}

class C extends A{
   void m(int a){ ... }
}

class D extends B{
   void m(int a){ ... }
}

class Main{
   void foo(String args[]){
      A a = new A();
     /* Some code */
     a.m(parameter);
   }
}
```

Listing 4.3: Example of virtual code after applying the zero profiler

```
class Main{
   void foo(String args[]){
      A a = new A();
     /* Some code */
     if(a instanceof B)
        ((B)a).m(parameter); /* Non-virtual call */
     else if(a instanceof D)
        ((D)a).m(parameter); /* Without-virtual call */
     else
        a.m(parameter); /* Virtual call */
   }
}
```

method `m` inside the `foo` procedure are: `A` ($frequency = 20\%$), `B` ($frequency = 20\%$), `C` ($frequency = 25\%$), `D` ($frequency = 20\%$) and `E` ($frequency = 5\%$)[3]. Suppose now that the type threshold is $Th_t = 20\%$ and the methods `B.m(int)` and `D.m(int)` have specialization hints. What we obtain at call site inside the method `foo` is shown in listing 4.3, where the operator `instanceof` returns true if the dynamic type of a variable is the type given as parameter.

4.3.2 First Profiler

As already mentioned the purpose of the first profiler is to understand if a given call site has been executed a sufficient number of times. To do so, it must be as small and light as possible: for this reason it should just use a counter that will be incremented each time the program flow reaches the call site. Once the counter reaches a given threshold (called *first profiler threshold*, Th_{First}) the first profiler has done his job.

One could argue that this method does not fit because it simply counts the number of times the call site has been reached without taking into account how many times the caller has been called itself. For example consider the following piece of code:

```
int main(){
    for(int i=0;i<100000000;i++){
        foo1(i,b,c);
    }
}

int foo1(int a, int b, int c){
    if(a<200)
        return foo2(b,c);
    else
        return foo3(c,d);
}
```

[3]Since we do not have a bound on the class hierarchy we cannot know a priori all possible dynamic types of an object, for this reason even a type that is not shown in the code above can appear.

```
int foo2(int b, int c){ ... }

int foo3(int c, int d){ ... }
```

Suppose that the threshold is 100 and that `foo2` and `foo3` are specializable. It is quite obvious that the method `foo3` will be execute many times more than `foo2`, so specializing foo2 would be useless.

Unfortunately, this way to proceed is quite costly because the specializer should also keep a global counter for each called method increasing the cost of profiling that we want to be as small as possible.

A possible solution that tries to merge the advantages of both the techniques mentioned is the following one: keep a counter for each call site; every time the caller method has been itself called, the counter of each call site has to be reduced by one unity, while when the control flow reaches a call site the counter has to be increased of two unities. Obviously, this way to proceed is more expansive than the previous one – there is at least a subtract instruction more. However, we will have better quality results.

This profiler can be actually run in conjunction with the zero profiler: if the specializer knows that a specializable virtual call has been reached a sufficient number of times, we can skip the first profiler injection inside the non-virtual calls generated when the type dispatcher will be built.

4.3.3 Second Profiler

The aim of the second profiler is to understand which values are constant like for a given parameter. As already said in the previous chapters, we can say that a variable (or a parameter) is *constant like* if it assumes most of the times the same value or a little range of different values. In practice we can say that a value is constant like for a given parameter if the parameter assumes that value with a frequency higher than a given frequency threshold. To find these values we can simply keep track of which values every specializable parameter assumes.

Before describing how the second profiler works, we have to define some useful values.

First of all the profiling procedure has to understand which parameters of the call instruction are subject to specialization. We will call the set of this parameter positions P. We can compute set P easily starting from all available expression hints for the call site:

$$P = \bigcup_{e \in expression} \bigcup_{subExpression \in e} \texttt{positionOf}(subExpression)$$

where `positionOf` is a function that returns the position of the given sub-expression, for example in the sub expression `p = #3 EQUALS 10`, we will obtain that $\texttt{positionOf}(p) = \{3\}$.

Now, consider that the specializer allocates for the the call site profiling B bytes of memory where to place the values of specializable parameters which index is contained in P. Assume also that there is a function $size(i)$ that, given the parameter at position i, it returns the size (in bytes) of parameter i implemented as follows:

$$size(i) = \begin{cases} 0 \text{ byte} & \textbf{if parameter at position } i \text{ is constant} \\ sizeof(INTEGER) \text{ byte} & \textbf{if parameter at position } i \text{ is an array} \\ sizeof(type(i)) \text{ byte} & \textbf{if parameter at position } i \text{ is a variable} \end{cases}$$

where $type(i)$ returns the type of parameter at position i. You could think that there is something wrong in the definition of the function: how could a parameter be constant? You should always remember that we are working at call site, so it may happen that due to some optimizations (for example constant propagation) we know that a parameter always assumes a single fixed value every time the function at call site is called. For this reason we can think that the "constant" parameter is already constant like. Moreover, the size of a specializable variable, that is an array, is the size of an integer variable, because, for arrays, we are going to detect if their size is constant like.

Then we can compute how much memory we use at each call of the method to store the actual values of the parameters. We call this value *block size* (K) and

Listing 4.4: Pseudo code for the second profiler implementation

```
for param in specializable parameter of call do:
    address = &memory + C * K + offset(param)
    *address = param
C = C + 1
if C > Th then reachedThreshold = True
C = C % Th
Call(param1, param2, param3,...  ,)
```

we can compute it as:

$$K = \sum_{p \in P} size(p)$$

Finally we can compute the threshold for the second profiler, that says how many blocks of K bytes can be stored in memory of size B bytes. Threshold Th can be computed as follows

$$Th = B \texttt{ div } K$$

where *div* is the integer division. This means that in the memory we can have at most Th different sets of values for specializable parameters that correspond to Th different calls.

Suppose now that there is a counter C that tracks how many blocks of values have already been injected. The pseudo code for this profiler is shown in listing 4.4.

The profiler just stores at the correct position the values for the specializable parameter inside the memory, and it tells the profiler that the memory is full once the threshold has been reached. The function `offset(i)` finds the offset of the parameter inside a block of values for a single call, and can be computed as follows:

$$offset(i) = \sum_{p \in P} \begin{cases} 0 \text{ bytes} & \textbf{if parameter at position } i \text{ is constant} \\ 0 \text{ bytes} & \textbf{if parameter at position } p \geq i \text{ is constant} \\ sizeof(type(p)) \text{ bytes} & \textbf{if parameter at position } p < i \text{ is constant} \end{cases}$$

Reaching the threshold doesn't mean that no more values could be put into the memory, it depends on the arrest of the profiler. If the profiling will be stopped

Listing 4.5: Example of code to profile by second profiler

```
m = {1,2,3,4,5,6,7,8,9,10}
n = {1,2,3,4,5}
for(i=0;i<1000;i++){
    a = i % 3;
    b = a + 1;
    v = (i%2)?m:n;
    call foo(a, b, v, i, 3);
}
```

after other many iterations, values will be overwritten.

Look at an example that better explains how this profiler works. Consider the piece of code shown in listing 4.5.

Suppose that parameters at positions one, three, four and five are integer variable parameters, that parameter two is a vector parameter and that parameter five is a constant parameter with value 3. Consider also that the callee method has the following hint expressions:

#1 STATIC AND #5 EQUALS 3
#1 STATIC AND #3 LENGTH STATIC
#3 LENGTH STATIC
#4 STATIC
#4 STATIC AND #5 STATIC
#1 EQUALS 1

Now we can compute the set P of specializable index of the method: $P = \{1, 3, 4, 5\}$, the value of the block size is $K =$ `sizeof(INTEGER)`$+$`sizeof(INTEGER)`$+$ `sizeof(INTEGER)` $+ 0 = 12$, with the assumption that `sizeof(INTEGER)` $= 4$. If we could have one hundred bytes of memory ($B = 100$) to use for this profiler, we can compute the threshold: $Th = 100/12 = 8$. For hypothesis consider also that the profiler will be stopped outside the `for` cycle. The result is shown in table 4.1.

Once the second profiler has collected all values, it has to understand which values are constant like and which are not. By definition a value is constant like if it has been seen with a frequency higher than a given threshold. Suppose

Table 4.1: Second profiler result for listing 4.5

K_i	1			2			3			4		
Index	1	3	4	1	3	4	1	3	4	1	3	4
	2	10	992	0	5	993	1	10	994	2	5	995

K_i	5			6			7			8		
Index	1	3	4	1	3	4	1	3	4	1	3	4
	0	10	996	1	5	997	2	10	998	0	5	999

to call this threshold Th_f. The computation of frequency should now take into account also the constant parameters, i.e. it has to consider all the values in the P set. Going on with the example started just above we can compute, for each parameter, every possible value with his frequency: results are shown in table 4.2.

Table 4.2: Second profiler frequency results for listing 4.5

Index	Value	Frequency
1	0	37.50%
	1	25.00%
	2	37.50%
3	5	50,00%
	10	50,00%
4	992	12.50%
	993	12.50%
	994	12.50%
	995	12.50%
	996	12.50%
	997	12.50%
	998	12.50%
	999	12.50%
5	3	100.00%

Depending on the value of threshold Th_f we can have different results; if we assume a very high threshold, for example $Th_f = 70\%$. Only the parameter at position five passes the test (with value 3) because it is a constant parameter. Therefore we can say that 10 is a constant like value for parameter three while the other parameters have no static values:

$static(1) = \{\}$
$static(3) = \{\}$
$static(4) = \{\}$
$static(5) = \{3\ [f\ =\ 100,00\%]\}$

If we reduce the frequency threshold value to $Th_f = 30\%$ we obtain again that the parameter at position five passes the test, the parameter at position 1 has two constant like values(0 and 2), and the parameter at position three has two constant like values (5 and 10) – which means that the length of the vector passed as parameter to the `foo` function assumes the found constant like values:
$static(1) = \{0\ [f\ =\ 37,50\%],\ 2\ [f\ =\ 37,50\%]\}$
$static(3) = \{5\ [f\ =\ 50,00\%],\ 10\ [f\ =\ 50,00\%]\}$
$static(4) = \{\}$
$static(5) = \{3\ [f\ =\ 100,00\%]\}$

We can, obviously, reduce again the value of the frequency threshold, and while we lower its value, many other values for the three parameters could become constant like. There is anyway a trade-off: if we lower the threshold we could obtain many *false constant like* values, i.e. values that the profiler thinks they are constant like while they are not. For example, if we put Th_f at a very low value, suppose $Th_f = 10\%$, we obtain that for parameter four the constant like values are $static(4) = \{992, 993, 994, 995, 996, 997, 998, 999\}$ which is obviously a wrong result. On the contrary, if the threshold frequency Th_f has a too high value, it will not find some true constant like values, as shown in the previous example when Th_f was put to 70%.

Not only the frequency threshold Th_f is an important setting for the second profiler, but also the size of the memory B, where actual values for parameters are stored, plays a key role and should be set very carefully. If B assumes a too low value it can collect too little values for each parameter, leading to an increase in the number of *false constant like* values. On the contrary, if the value

of the second profiler variable B is too high, the second profiler could decrease the performance of the program because it will be executed many (possibly useless) times, consuming also a large amount of memory space. In this last case, anyway, the probability to obtain a *false constant like* value is much lower than the previous one.

At this point, the second profiler has ended his job and sends the list of constant like values of each parameters (if any) to the dispatcher. If no constant like values for specializable parameters have been found, the specializer sets this call site as un-specializable and will not consider it again.

4.4 Callee Specialization and Dispatcher Injection

This phase can be divided in three sub-phases: *constant like expression detection*, *callee method specialization* and *dispatcher injection*.

4.4.1 Constant Like Expressions Detection

Before injecting the actual dispatcher, the specializer have to understand which expression hints are useful to generate a specialized method and with which values. To perform this task the specializer requires the output of the second profiler phase. At this point, all that we know is a list of constant like values the specialized parameters have and the hints. What we have to compute at this point are *constant like expression*s: a constant like expression is an expression where each sub-expression has a fixed value for its parameter and its whole frequency is higher than the frequency threshold Th_f. Generating all the possible constant like expressions is quite easy.

First of all the specializer has to generate all possible combinations of values for the sub expressions; if a sub-expression has type `EQUALS` that sub expression can assume only that value if and only if that value is also constant like for

the parameter associated at the given expression. To compute the frequency of the expression of a given combination, the specializer simply has to multiply the frequency of the values of each sub expression[4]. If a sub-expression has no constant like values (because no constant like values has been found by the second profiler or because it is an `EQUALS` sub-expression but its comparison value is not constant like) it cannot be considered as constant like.

Here there is a simple example. Consider again the listing 4.5 with his associated hint expressions and the constant like values when $Th_f = 30.00\%$. For all the hint expressions we will show all the possible combination of values, highlighting which expressions are actual constant like and which are not. The results are shown in table 4.3.

Table 4.3: Second profiler static expression for listing 4.5

Expression	Value combination				Static
	Par. 1	Par. 3	Par. 4	Par. 5	
`#1 STATIC AND #5 EQUALS 3`	0			3	Yes [f = 37.50%]
	2			3	Yes [f = 37.50%]
`#1 STATIC AND #3 LENGTH STATIC`	0	5			No [f = 18.75%]
	0	10			No [f = 18.75%]
	2	5			No [f = 18.75%]
	2	10			No [f = 18.75%]
`#3 LENGTH STATIC`		5			Yes [f = 50,00%]
		10			Yes [f = 50,00%]
`#4 STATIC`			??		No
`#4 STATIC AND #5 STATIC`			??	3	No
`#1 EQUALS 1`	1				No [f = 25,00%]

Constant Like Expressions Merge

Sometimes it could happen that two different expressions could possibly generate the same constant like expression. Consider for example the following two constant

[4]Here we assume that parameters are independent. This cannot always be true. For example, suppose that we have a specializable method `foo(int a, int b)` and that we can specialize on both `a` and `b`: if at a generic call site we have that `foo` is called as `foo(variable, variable+1)`, it is quite obvious that `a` and `b` are not independent. Anyway, since there are no other information about parameter independence, we have to assume that they are independent.

like expressions:

```
(1) #1 STATIC
                Values : [#1=0]

(2) #1 EQUALS 0
                Values : [#1=0]
```

As we will see next in this chapter, both these constant like expressions generate the same specialized method. For this reason we can safely remove one of these constant like expressions.

More precisely, we can say that two constant like expressions are *redundant*, if they have the same set of parameters position P, and for each element of the set, each position will assume the same value. Therefore, if two constant like expressions are redundant, we can remove one of them without compromising the final result of specialization.

4.4.2 Callee Method Specialization

At this point the specializer has collected a list of constant like expressions useful to generate specialized versions of the original callee method: every constant like expression generates a different specialized method.

The development of a specialized method is quite a simple step because we know the constant like expression. What we have to do is generate a new method with the same code of the callee method but with a different signature. The signature of the new method is just like the signature of the original callee without the parameters that have been specialized and that now have their own value. For this reason, having removed those parameters, we have to reintroduce them again in the code, initializing them with the value associated as described by the constant like expression. Actually this cannot be done in case of sub-expressions where the action is `LENGTH STATIC` because what we have specialized here is the size of the vector. To specialize a callee method with an expression of such a kind, we have not to remove the associated parameter, instead we have to replace every

Listing 4.6: Example of method to be specialized knwing its static expression

```
int foo(int a, int b, int c[], int d){
    int k = 0;
    if(a == 0)
        for(int i=0;i<c.length;i++){
            k = k + c[i];
        }
    else if(a == 1)
        for(int i=0;i < b; i++){
            k = k + d;
        }
    return k;
}
```

attempt to read the length of the vector with the fixed found value. Obviously, this can be done only if the specializer is sure that we are actually reading the value of the passed parameter and that the vector length cannot be changed anywhere in the code by some vector aliases.

Consider a simple but complete example. Observe the function `foo` shown in listing 4.6 at page 96. After the profiler phase the dispatcher knows its constant like expressions:

```
(1) #1 EQUALS 0 AND #3 LENGTH STATIC
                Values : [#1=0,#3=5]

(2) #1 EQUALS 1 AND #2 STATIC AND #4 STATIC
                Values : [#1=1,#2=10,#5=5]
```

For the first constant like expression we obtain a new method as shown in listing 4.7. As it has been said above, the vector parameter `c` has not been deleted from the signature, only the first `a` parameter has been removed. At this point, for this new method, the specializer procedure has completed his job, and the method is sent to the recompile routine that will heavily optimize the code producing something like the resulting code shown in listing 4.9.

For the second constant like expression, we obtain a new method, different from the previous one, as shown in listing 4.8, that will be next specialized resulting in

Listing 4.7: Method shown in listing 4.6 specialized with the first static expression

```
int foo(int b, int c[], int d){
    a = 0;
    c.length = 5;
    int k = 0;
    if(a == 0)
        for(int i=0;i<c.length;i++){
            k = k + c[i];
        }
    else if(a == 1)
        for(int i=0;i < b; i++){
            k = k + d;
        }
    return k;
}
```

Listing 4.8: Method shown in listing 4.6 specialized with the second static expression

```
int foo(int c[]){
    a = 1;
    b = 10;
    d = 5;
    int k = 0;
    if(a == 0)
        for(int i=0;i<c.length;i++){
            k = k + c[i];
        }
    else if(a == 1)
        for(int i=0;i < b; i++){
            k = k + d;
        }
    return k;
}
```

Listing 4.9: Method shown in listing 4.6 specialized and optimized with the first static expression

```
int foo(int b, int c[], int d){
    return c[0]+c[1]+c[2]+c[3]+c[4]+c[5];
}
```

Listing 4.10: Method shown in listing 4.6 specialized and optimized with the second static expression

```
int foo(int c[]){
    return 50;
}
```

Listing 4.11: Dispatching of method shown in listing 4.6 for given static expressions

```
if(a==0 && c.length==5)
    foo(b,c,d);
else if(a == 1 && b == 10 && d == 5)
    foo(c);
else
    foo(a,b,c,d);
```

the method shown in listing 4.10

4.4.3 Dispatcher Injection

Now we have a list of constant like expressions, associated with a specialized callee method. At the given call site in the caller method we can substitute that call with a dispatcher, that selects between the specialized callee methods and the original callee method. Even this step is quite easy to execute: for each constant like expression, it generates an `if` statement that checks if the actual parameters match the required values for each sub expression, if so it calls the associated specialized method. An `else` statement must also be injected to ensure that if no specialized method can be called, the original method will be.

Have a look to a simple example. Consider the constant like expressions and their associated method shown in the previous section 4.4.2. For each expression it injects the `if` statement and at the end it injects the `else` statement. The result is shown in listing 4.11.

Dispatcher Removal

Sometimes it could happen that the program behaviour changes (see section 4.5.1) and that the resulting specialized method will not be called any more, leading to a loss of performance caused by the useless dispatcher injected. For this reason it should be very useful to understand when such a change happens. To solve this problem we can simply introduce a counter at this call site: this counter will be decremented of one unity every time we call a specialized version of the method, and it will be increment of two unities when the original method will be called. If the counter reaches a given threshold the dispatcher can be removed and the original call can be restored.

At this point, two different behaviour policies are feasible. In the first policy we can mark the specialized call site as un-specializable and never take care of it. In the second policy we can set the call site as clean and restart the profiling process. Actually, there is a third policy that is the mix of the previous two: the specializer marks the call site as un-specializable but after some times it could decide to re-mark the call site as clean and execute again the profiling process. Anyway this third policy, even if it is the most complete, is much more complex and heavy to implement and execute inside a JIT compiler.

Constant Like Expression Ordering

The injection of the `if` statement in the dispatcher should be done in a suitable order. For example, suppose that you have the following constant like expressions for method shown in listing 4.6:

```
(1) #1 EQUALS 0
                Values : [#1=0]

(2) #1 EQUALS 0 AND #2 EQUALS 3
                Values : [#1=0,#2=3]
```

If we inject the `if` statements for these two constant like expressions in the given order, we obtain the code shown in listing 4.12. With a simple analysis we

Listing 4.12: Dispatching of specializable method foo in the wrong way

```
if(a == 0)
    foo(b,c,d);
else if(a == 0 && b == 3)
    foo(c,d);
else
    foo(a,b,c,d);
```

Listing 4.13: Dispatching of specializable method foo in the correct way

```
if(a == 0 && b == 3)
    foo(c,d);
else if(a==0)
    foo(b,c,d);
else
    foo(a,b,c,d);
```

Listing 4.14: Dispatching of specializable method foo: a possible good way

```
if(a == 0 && b != 3)
    foo(b,c,d);
else if(a == 0 && b == 3)
    foo(c,d);
else
    foo(a,b,c,d);
```

find out that the second `if` statement – `if(a == 0 && b == 3)` – will never be executed because, even if `a==0` and `b==3` the first `if` statement will be executed. What we actually want is something similar to what is shown in listing 4.13.

There are substantially two ways to obtain this behaviour. The first one is injecting the `if` statement in the given order of the constant like expression, but when injecting the first profiler we have also to require that $b \neq 3$. The result is shown in listing 4.14.

The second way requires to order the constant like expressions to obtain the correct behaviour. To obtain this behaviour we can automatically order the constant like expressions, injecting first the `if` statements generated from constant like expressions with more sub-expressions. Otherwise the specializer can require that the constant like expressions will be already given in the correct order.

Since we have already done the hypothesis that the expression hints for each method will be generated off-line by an external tool, we can require that the constant like expression ordering would be done by the same tool. This will simplify further the partial evaluation mechanism.

4.5 Open Issues And Possible Solutions

In this chapter it has been shown a quite complete, but enough simple to be implemented inside a JIT compiler, specialization process. Anyway, there are some cases that cannot actually be covered, due to the complexity of the problem or the great variability of the execution of a generic program.

4.5.1 Alternate Execution Phases

As already mentioned, a program execution can have different phases of execution, involving different values that have to be passed to the specializable callee method at a given call site. This behaviour could arise for many reason. For example, consider a generic server process: during the day time it can receive a lot of

requests from different users making the specialization quite difficult because of the different values it can receive in input.

On the contrary, in the night it receives less requests from the same group of sleepless people, so the execution process can be widely specialized. Obviously this behaviour happens every day so the specializer process has to continue specializing and un-specializing each call site leading to possible lost of performance. Furthermore nobody can say to the specializer that each night we can specialize each call site in the same way as the previous ones, so previous specialized method could be useless and just occupy memory.

This is therefore a practical proof that in general partial evaluation should be carefully tuned in order to obtain gain applying it.

4.5.2 Specialization Cascade

It could happen that we want specialize a call site that calls a method that itself contains some specializable call sites. What said in this chapter obviously applies also in this case.

Problems arise when in the first caller method we generate the specialized method: even these methods can contain calls to other specializable methods. We can simply treat these specialized method as standard one but we have to decide whether we want to keep or discard all previous collected data.

4.5.3 A possible solution: Specialization window growing

To partially avoid all the disadvantages described above, we could think of enlarging the *specialization window*. The specialization window describes how many call sites we actually consider in order to specialize just one call site. The specialization procedure described in this chapter has implicitly set the specialization window to one: only the call site that can be specialized is considered. To gain better results we could think of not only considering this call site but also the call site where the caller of the specializable callee at the given call site has been

Listing 4.15: Example code for window growing

```
void main(){
    for(int i=0; i < 1000; i++){
        foo1();
        foo2();
    }
}

void foo1(){
    for(int i = 0; i < 5000; i++)
        foo3(i % 1000);
}

void foo2(){
    for(int i = 0; i < 5000; i++)
        foo3(0);
}

void foo3(int a){
  foo(a, a+1);
}

void foo(int a, int b){
    if(a == 0)
        print 5000 * b;
    else
        print 0;
}
```

called itself. In this case the specialization window has size two.

Enlarging the specialization window, we have more opportunity to select if this method should be specialized or not. Consider for example listing 4.15. Suppose that method `foo` can be specialized if we know that `#1 EQUALS 0 AND #2 STATIC` and that `#1 EQUALS 1`, so we could inject profilers and dispatchers at call site inside method `foo3`. If you take a deep look at the code, you can see that if `foo3` is called by `foo1`, no specialization is possible, while if `foo3` is called by `foo2` the specialization can actually take place. If we consider a specialization window with size one, we should continue specializing and un-specializing the call site of `foo`

inside method `foo3`. On the contrary, if we increase the specialization window with value 2, we can see that specialization is possible only if `foo3` is called by `foo2`. Unfortunately there are two main problems with this technique: first of all it is difficult to understand the size of the specialization window, and it is very heavy to perform because it would involve the reading of the execution call stack, that it is a very time consuming operation, infeasible in a dynamic compiler.

In this thesis I am not going to develop also this technique due to its complexity and because it is quite a time consuming approach.

Chapter 5

System implementation

This chapter will explain how the specialization techniques described in chapter 4 inside the ILDJIT compiler have been implemented.

This compiler has been chosen for implementation because, even if it is a quite complete tool, it is not as complex as other tools like Mono and PNet. ILDJIT has also a very large number of worthwhile optimizations that could possibly be very useful during the specialization phase. At last ILDJIT has also been released under the GPL license so its source code is freely available as already mentioned. For all these reasons, ILDJIT will be the base on which a specialization module will be built.

5.1 ILDJIT Background

First of all, to better explain the development of the specializer, it is necessary to discuss briefly the ILDJIT compiler software structure.

5.1.1 ILDJIT Modules

The ILDJIT compiler has been splitted in different modules. Each module exposes one or more interfaces and different data structures to other packages. Anyway, since the compiler has been written in C, packages of lower level cannot see the data structures presented in the higher level. This could lead to problems that

can be solved using some tricks.

The current developed modules are:

- **libiljitu** This module contains various tools useful for all the other modules, for example it contains the descriptions of all the available C types[1] and a description of the IR language;
- **libcompilermemorymanager** This package administrate the dynamic memory needed by the compiler, but it does not for the running application;
- **libiljitsm** This module implements the section manager for CIL files;
- **libiljitmm** This module manages the meta-data manager for CIL files;
- **libiljitir** This module contains the description of an IR method, i.e. the method represented in the Intermediate Representation fashion (See 5.1.2 for more details about IR method structure);
- **libiljitiroptimizer** This module describes the optimization framework. This package enable ildjit to optimize methods with many different kinds of optimizations, that can be loaded dynamically;
- **libiljitirprofiler** This module enables the profiling of the code for the ildjit compiler; since the partial evaluation process presented since here uses some different profilers, it has been decided to modify this module in order to enable `iljit` to perform specialization in methods;
- **libdla** This module enable the activation of the DLA dynamic compilation technique (see section 3.1.1) inside the compiler;
- **iljit** This is the core module of the project, it substantially contains the virtual machine of the compiler. Here the interface for the IL methods, i.e.

[1] See file `libiljitu/src/jitsystem.h`.

for the methods represented in the CIL Intermediate Language fashion, is also described (See 5.1.2 for more details about CIL method);

- **iljit-ecmasoft-decoder** This module contains the implementation for the CIL files decoder;

- **Garbage Collectors** ILDJIT enables you to select which garbage collector to use. At present there are four different garbage collectors: the *shifter* garbage collector, the *mark and sweep* garbage collector, the *copy* garbage collector and the *bwm* garbage collector (which is a wrapper for the garbage collector written by Boehm, Demers and Weiser [38]);

- **Optimizers** ILDJIT supports a various number of program analysis and optimizations. All these analysis and optimization are implemented as external plugins. There are various number of it that can do a lot of common operation: basic block identification, bounds check removal, branch prediction, constant propagation, copy propagation, dead code elimination, escapes analysis, liveness analysis, loop identification, loop invariant identification, array bounds check removal, null check removal, reaching definitions analysis, pre and post dominators analysis and trace scheduling.

5.1.2 CIL Method, IR Method and Compiled Method

A method loaded from a generic CIL file (either an executable program file or a shared DDL library) can be found in the three formats: CIL, IR and machine code.

The CIL format is the description of the method[2] in the original bytecode format. This is the highest level language available. As already mentioned, this is a stack-based language, so to be executed it must be translated in a register-based

[2]See files `iljit/src/ilmethod.[h,c]`

language, i.e. the IR language. This process is performed by the `iljit` module[3].

IR is a register-based language, with a more low level vision of the code respect the CIL language. It is quite similar to the assembly language, but with much more features. This language is suitable for the optimization process since it abstracts most of the complexity of the machine code language, but it is not as complex as higher languages such as CIL. For this reasons, every code manipulation (either executed by one of the optimization plugins or by the specialization process) involves the modification of the method at this level. For each structure that represents an IL method there is one and only one associated IR method[4]. This condition will be lost when a specialized IR method will be introduced, when required, for the CIL method. IR is a strongly typed language, each instruction has its own fixed type; the main types are: `IRUINT8`, `IRUINT16`, `IRUINT32`, `IRUINT64`, `IRNUINT`,`IRINT8`, `IRINT16`, `IRINT32`, `IRINT64`, `IRNINT`, `IRFLOAT32` and `IRFLOAT64`. This language offers a lot of simple instructions of many kinds[5], some of such instructions, that will be used to build the profiler and the specializers, are:

- **Math Instructions**: `IRADD`, `IRMUL`, `IRSUB`, `IRDIV`, `IRREM`, ...;
- **Boolean Instructions**: `IRAND`, `IRNOT`, `IROR`, `IRXOR`, ...;
- **Comparison Instructions**: `IRLT`, `IREQ`, `IRGT`, ...;
- **Branch And Label Instructions**: `IRLABEL`, `IRBRANCHIF`, `IRBRANCHIFNOT`, `IRBRANCH`, ...;
- **Method Call Instructions**: `IRCALL`, `IRVCALL` (for virtual calls), `IRICALL` (for calls to native methods), ...;

[3]See method `translate_method_from_cil_to_ir` in files `iljit/src/cil_ir_translator.[h,c]`.

[4]See files `libiljitir/src/ir_method.[h,c]`.

[5]For a complete description see file `libiljitu/src/ir_language.h` that presents the IR language grammar.

- **Memory Access Instructions**: `IRLOADREL`, `IRSTOREREL`, ...;
- **Variable Assignment Instructions**: `IRLOAD`, `IRSTORE`, ...;

In order to be executed, a method must be finally translated into machine code. This process is quite simple, starting from the IR method represented[6]. Inside ILDJIT this last translation phase is made possible by the use of the `libjit` library [39], that is part of the Portable .NET project.

A generic method representation can assume one of the following states[7]:

- `METHOD STATE JITFUNCTION NOT CREATED`: The CIL method data structure has been created but the compiled jit function does not;
- `CIL STATE`: The method has been loaded in CIL format;
- `MACHINE CODE STATE`: The method has been implemented directly as machine code;
- `NEEDS RECOMPILE STATE`: The method needs to be recompiled before next executions;
- `IR STATE`: The method has been translated from CIL to IR language;
- `EXECUTABLE STATE`: The methods has been translated from IR to machine language.

5.2 Specialization Implementation Technique

In this section how the complete specializer has been implemented inside the `libiljitirprofiler` module will be described. The specialization phase has followed the classical specialization process. Figure 5.1 shows how the specialization process works inside the ILDJIT compiler.

[6]See method `make_jit_method` in files `iljit/src/ir_jit_translator.[h,c]`.
[7]See file `iljit/src/system_manager.h`

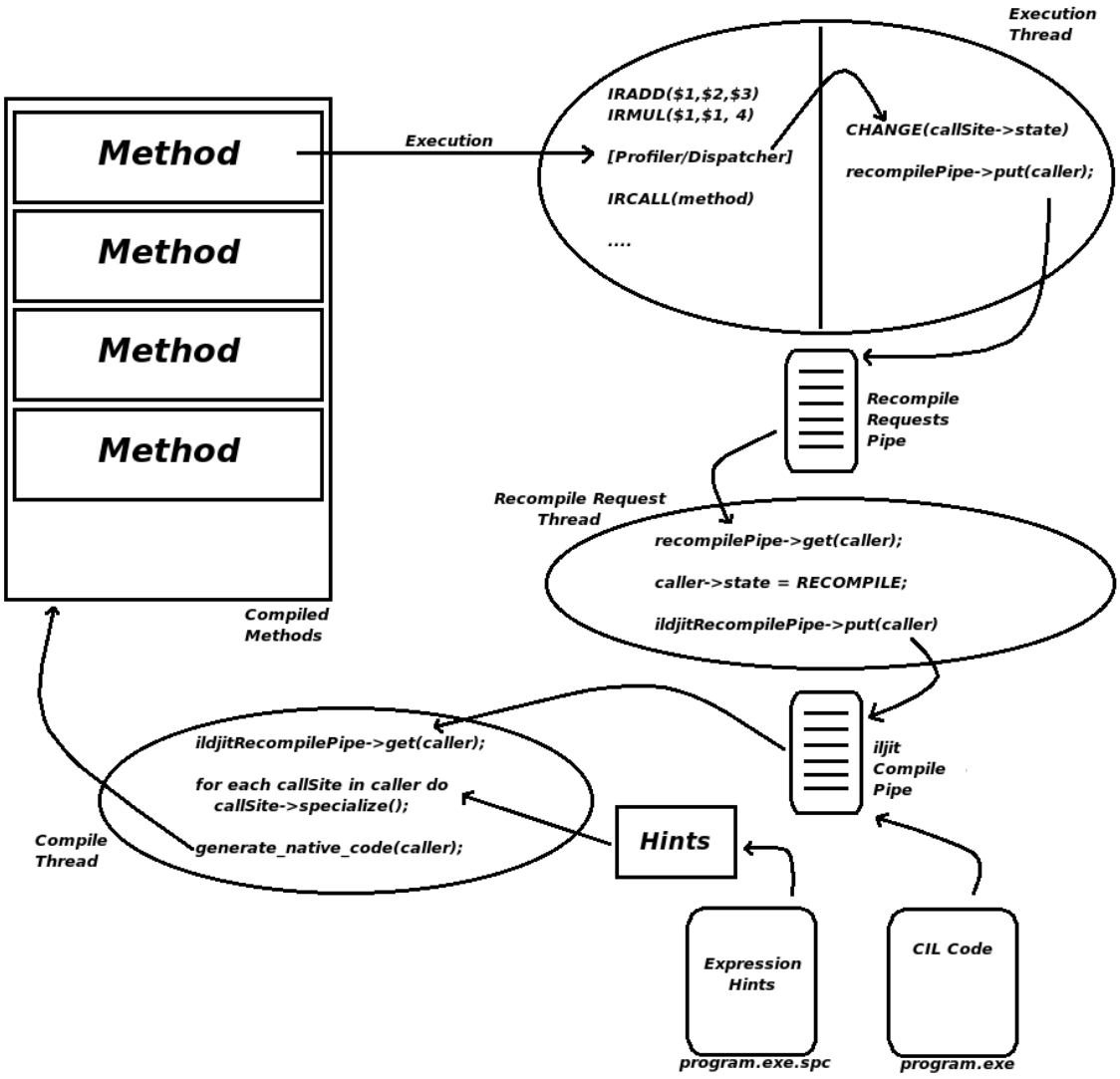

Figure 5.1: The specialization execution model inside iljit

First of all in the preprocessing phase all the specialization expression hints are loaded from an external file[8]. This phase takes place in the bootstrap phase of the iljit compiler where also all the required specialization structure will be created and initialized.

Each method loaded from the program file (or from an external libraries) in CIL format must me translated from CIL language to IR language, and from IR language to native compiled code. While the process of conversion between IR and native code takes place in the compile thread, each call site is checked in order to understand if it can be specialized or if it is already been specialized or, eventually, if it needs a change of state (for example suppose that the call site is in the state `CLEAN`, the specializer must inject the first profiler instructions and move the call site to the state `FIRST PROFILER INJECTED`). After this passage we have an IR method where each call site is in the correct state.

Eventually, the method compiled will be executed by the *execution thread*. It may happen that a call site needs to change its state because a threshold has been reached (for example, in the first profiler, the call site has been called a sufficient number of times). When this happens a native call (i.e. a call hard coded inside that compiler and that has been written in C) will be called; this call will change the state of the call site to a new state depending on the current call site state, for example if the call site state is `SECOND PROFILER INJECTED` the state will be moved to `DISPATCHER REQUEST`. After this, the method will placed in the *recompile request pipe*, because it needs to be recompiled again to change the call site that has changed its state.

The method that needs to be recompiled is now in the recompile request pipe, and it will be removed from this pipe from the *recompile request thread*. This is a very simple thread: it just changes the state of the IL method to `IR_STATE`

[8]The name of this file (if present) is obtained joining the name of the file to execute with the string ".spc", for example, the program "foo.exe" has associated the specialization file "foo.exe.spc"

state and place it inside the *ildjit compile queue*. Changing the state for the IL method is necessary for the recompile process in order to take place for the method injected inside the compile queue. Moreover, the new IL method state requires only to translate the method from IR code to machine code, avoiding the cost of the translation from CIL code to IR code.

Finally, this method will be removed from this queue by the *compile thread*, that restarts the compile process for this method.

The most of the work is done by the compile thread that for each call site must decide what to do. This is chosen depending on the call site state as described in section 4.1:

- `SPECIALIZER_CALLSITE_STATUS_CLEAN`: The call site is clean. This means that the first profiler must be injected inside the code, around the given call site;
- `SPECIALIZER_CALLSITE_STATUS_FIRST_PROFILER_INJECTED`: The first profiler has been injected in the call site;
- `SPECIALIZER_CALLSITE_STATUS_SECOND_PROFILER_REQUEST`: The first profiler has reached his threshold and it must be replaced by the second profiler;
- `SPECIALIZER_CALLSITE_STATUS_SECOND_PROFILER_INJECTED`: The second profiler has been injected in the call site;
- `SPECIALIZER_CALLSITE_STATUS_DISPATCHER_REQUEST`: The second profiler has terminated his job, collecting enough information about the actual parameters of the callee method, so the second profiler must be removed and the dispatcher must be injected;
- `SPECIALIZER_CALLSITE_STATUS_DISPATCHER_INJECTED`: The dispatcher has been injected in the call site;

- `SPECIALIZER_CALLSITE_STATUS_DISPATCHER_REMOVE`: After some iterations, the specializer founds that keeping this dispatcher leads to a lost of performance, so the dispatcher must be removed, and the first profiler must be injected again in the same call site;

- `SPECIALIZER_CALLSITE_STATUS_DISPATCHER_UNSPECIALIZABLE`: The call site is unspecializable; this happens, for example, when a new call site is injected inside a dispatcher;

- `SPECIALIZER_CALLSITE_STATUS_TYPE_DISPATCHER_REQUEST`: After the first profiler, if in the call site there is a virtual call then the specializer must remove the first profiler and inject the type dispatcher;

- `SPECIALIZER_CALLSITE_STATUS_TYPE_DISPATCHER_INJECTED`: The type dispatcher has been injected in this call site;

- `SPECIALIZER_CALLSITE_STATUS_TYPE_DISPATCHER_REMOVE`: After some iterations, the specializer founds that keeping this type dispatcher leads to a lost of performance, so the type dispatcher must be removed and the first profiler must be injected again in the same call site;

Many policies can be chosen to change the state of a generic call site. We will follow what described in chapter 4: to better track the program behaviour, if a call site must be un-dispatched, or the zero profiler or the second profiler don't find suitable values, the call site will return to the `CLEAN` state.

Anyway, there is a slightly difference between the implementation of partial evaluation described in this chapter and what has been described in chapter 4: while in the previous chapter zero profiler and first profiler are two distinct entities, here they have been merged in a single entity to gain performance.

Summarizing, the phases involved in the method specialization are: *bootstrap and preprocessing*, *specializable call site detection*, *zero and first profiling*, *second*

Listing 5.1: The `ir_specializer_t` structure

```
typedef struct ir_specializer_t{
    XanList         *hintsForMethods;
    XanList         *callSitesForMethods;
    XanList         *wrapperForMethods;
    XanList         *specializedMethods;

    XanPipe         *recompileRequestPipe;
    pthread_t       recompileRequestThread;

    void            *pipeline;
    JITUINT32       pipelinePriority;
} ir_specializer_t;
```

profiling, *constant like expression detection*, *callee method specialization*, *dispatching* and *de-specialization*.

5.2.1 Bootstrap And Preprocessing

In this phase the specializer data structure will be initialized and the expression hints loaded from file.

Specialization Initialization

First of all we have to initialize a structure that will be used in the specialization process. Such a structure aims to track: all call sites for each method, all available hints for each method, all methods wrapper and all the specialized methods. The structure with this purpose, that has been called `ir_specializer_t`[9], is shown in listing 5.1.

The four lists `callSitesForMethods`, `hintsForMethods`, `wrapperForMethods`, `specializedMethods` are containers for the required information described above. The `recompileRequestPipe` is the queue used by the specializer to ask for the recompilation of the method, while the thread variable `recompileRequestThread` is the ID of the recompile request thread.

[9] see files `libiljitirprofiler/src/irprofiler.h` and `libiljitirprofiler/src/irspecializer.c`

Listing 5.2: The `specializer_method_wrapper_t` structure

```
typedef struct specializer_method_wrapper_t{
   void          *ilMethod;
   JITUINT32     recompileStateValue;

   char*         (*_getName)(void* ilMethod);
   char*         (*_getFullName)(void* ilMethod);
   char*         (*_getSignature)(void* ilMethod);
   ir_method_t*  (*_getIRMethod)(void* ilMethod);
   void          (*_setState)
                        (void* ilMethod, JITUINT32 state);
   JITUINT32     (*_getState)(void* ilMethod);
   void          (*_setJITFunction)
                        (void *ilMethod, void* jitFunction);
   void*         (*_getJITFunction) (void *ilMethod);
} specializer_method_wrapper_t;
```

As mentioned above, for each CIL method we have a *method wrapper*, which is described in listing 5.2. This data structure has been created since we do not have complete access to the CIL method data structure `_ILMethod`[10], because it is at an higher level of the ILDJIT modules definition and we can only "know" it as a void pointer, so there was the need to have an access to an IL method as simple as possible. As you can see, inside the `specializer_method_wrapper_t` structure we have the pointer to the IL method and we also have a list of function pointers that ensure the access to the IL method structure.

Load expression hints from files and metadata

Once the specializer has been initialized, it can load the specialization hints from file and, possibly, from method metadata. Unfortunately, since, at present, there are no "friendly" compilers which can write metadata with specialization hints, they will be read only from file. The read file has a quite simple structure. For each method that the user thinks could be worthwhile specializing, he has simply to define:

[10] See file `iljit/src/ilmethod.h`.

Listing 5.3: The `specializer_expression_hint_t` structure

```
typedef struct specializer_expression_hint_t{
    JITUINT32               position;
    JITUINT32               action;
    union{
        JITINT64            ivalue;
        JITFLOAT64          fvalue;
    }                       value;
    JITUINT32               depth;
    struct specializer_expression_hint_t * next;
    struct specializer_expression_hint_t * prev;
} specializer_expression_hint_t;
```

- The full name of the method (for example `System.String.CompareTo`). As shown in the the example, inside the name, the user has also to place the exact namespace definition in order to distinguish between different methods with the same name;
- The complete signature of the method as generated by iljit (for example `System.Int32 CompareTo(System.String)`);
- The zero profiler constant like type threshold and memory size, that can be used to store the data type values, the first profiler threshold, the size of the memory for the second profiler, the value of the frequency threshold for the second profiler, the threshold for the dispatcher and for the type dispatcher removal process;
- A list of expressions that describe the specialization hints as described in section 4.2;

To perform this operation there is the need, first of all, of a structure that describes an expression hint (that can eventually be joined with the AND boolean operator to other expressions). This structure is called `specializer_expression_hint_t` and is shown in listing 5.3. The `specializer_expression_hint_t` structure describes, for an expression, the parameter to which it refers with the `position`

Listing 5.4: The `specializer_hintsForMethod_t` structure

```
typedef struct specializer_hintsForMethod_t{
    char          *methodCompleteName;
    char          *methodSignature;
    JITFLOAT32    zeroProfilerTypeThreshold;
    JITUINT32     zeroProfilerMemorySize;
    JITUINT32     zeroDispatcherRemoveThreshold;
    JITUINT32     firstProfilerThreshold;
    JITUINT32     secondProfilerMemorySize;
    JITFLOAT32    secondProfilerStaticThreshold;
    JITUINT32     dispatcherRemoveThreshold;
    XanList       *expressions;
} specializer_hintsForMethod_t;
```

field, the action that the specializer has to perform on the given parameter[11] (i.e. it can be `STATIC`, `EQUALS`, `LESS_THEN`, ...) and the comparison value of the action, if required (union field `value`).

We need also a structure that describes the whole list of expressions for a given method, and that also describes all the profilers and dispatchers parameters (thresholds and memory sizes). This structure is called `specializer_hintsForMethod_t` and is shown in listing 5.4.

5.2.2 Specializable Call Sites Detection

Once the specializer has loaded the specialization expressions, while the compiler translates a method from the IR language to the executable code, for each each call site (i.e. each instruction[12] with type `IRCALL` or `IRVCALL`) it detects if it is specializable. This quite trivial task is performed by the function `manageCallSiteBeforeCompilation` and works as follows: the specializer procedure straightforwardly detect the method called at the given call site and it checks if its name (including the name space definition) and its signature appears in the list of the specializable methods or if

[11] See file `libiljitirprofiler/src/irspecializer.h`

[12] See instruction structure definition `t_ir_instruction` inside file `libiljitir/src/ir_method.h`

the method metadata coin tans some information about the specialization of the method itself. If this happens, from the list of call site of each method it retrieves the call site structure, and on it performs the required operations depending on the state. Here you can see the pseudo code for this operation:

```
void manageCallSites(ir_method_t* method){
    instruction = method->firstInstruction;
    while(instruction != NULL){
        if(instruction->type == IRCALL ||
            instruction->type == IRVCALL){
            hints = getHints(method->name,
                             method->signature, method->metadata);
            if(hints != NULL){
                callSite = getCallSite(method, instruction);
                callSite->performAction(callSite->status, hints);
            }
        }
        instruction = method->nextInstruction(instruction);
    }
}
```

The call site structure (`specializer_method_callSite_t`) is shown in listing 5.5. As you can see, this structure contains a wrapper for the called method, the instruction this call site refers to, the state, the hints associated to the called method, the instructions added by the profilers or by the dispatchers and the profilers and dispatchers data structures that will be presented next in this chapter.

5.2.3 Zero And First Profiling

As already mentioned in this chapter, the zero and the first profiler has been merged together, so, actually, there is only one data structure that describes both these profilers.

This structure has been named `specializer_method_callSite_firstProfiler_t` – see listing 5.6 – and it describes: the counter needed for both first and zero profiler, the threshold reached flag and the memory used to store data type values for the zero profiler. Moreover, this structure contains a pointer to the call site con-

Listing 5.5: The specializer_method_callSite_t structure

```
typedef struct specializer_method_callSite_t{
    specializer_method_wrapper_t *methodWrapper;
    t_ir_instruction      *call;
    JITUINT32     status;
    specializer_hintsForMethod_t *hints;
    XanList       *addedInstructions;

    specializer_method_callSite_firstProfiler_t  *firstProfiler;
    specializer_method_callSite_secondProfiler_t *secondProfiler;
    specializer_method_callSite_dispatcher_t     *dispatcher;
    specializer_method_callSite_type_dispatcher_t*typeDispatcher;
} specializer_method_callSite_t;
```

Listing 5.6: The specializer_method_callSite_firstProfiler_t structure

```
typedef struct specializer_method_callSite_firstProfiler_t{
    JITUINT32     counter;
    JITUINT32     thresholdReached;
    JITUINT32     *memory;
    void          *callSiteContext;
} specializer_method_callSite_firstProfiler_t;
```

text, i.e. a pointer to the specializer_method_callSite_t structure that owns this profiler.

Once a call site reaches the SPECIALIZER_CALLSITE_STATUS_CLEAN state, the first profiler must be injected. As already mentioned in the previous chapter many different implementations are possible. Here two of them will proposed and discussed.

First Implementation

The first implementation model of the first profiler is the most simple and immediate. Here, at each call site, the specializer simply injects the following IR code[13]

[13]The symbol [variable] refers to the address of variable. The variable contained inside the parenthesis can be either a generic IR variable or a variable of the profiler/dispatcher. Moreover, the notation $N indicates a generic variable, not necessary a variable with N as identifier: the number is simply used to have a straight way to refer to the same variable inside the code. When

(if the call is not a virtual call):

```
$0 = IRLOADREL [thresholdReached], 0
IRBRANCHIF $0, #Label
$0 = IRLOADREL [counter], 0
$0 = IRADD $0, 1
IRSTOREREL $0, [counter], 0
$0 = IRLT $0, THRESHOLD_FIRST_PROFILER
IRBRANCHIF $0, #Label
IRICALL thresholdReached(callSite, method)
IRLABEL #Label
IRCALL Method, Parameters
```

This code means that just before the call itself, the specialization procedure has to inject instructions to load the first profiler counter and the first profiler threshold reached flag (using the `IRLOADREL` instruction). If the threshold has already been reached (i.e. the threshold reached flag is true), we can skip the increment of the counter. If this does not happen, the counter must be incremented and, if its value exceeds the threshold for the first profiler the native method `thresholdReached` must be called, in order to put the method inside the recompile request queue.

For a virtual call we have to use a different approach, because we have to inject more instructions in order to track the actual types of the object that calls the method. This is the IR code that must be injected for a generic virtual call:

```
$0 = IRLOADREL [thresholdReached], 0
IRBRANCHIF $0, #Label
$0 = IRLOADREL [counter], 0
$0 = IRADD $0, 1

IRBRANCHIFNOT param0, #Label
$2 = IRLOADREL param0, layoutOffset
$1 = IRREM $0, ZERO_PROFILER_MEMORY_SIZE
$1 = IRMUL $1, sizeof(JITUINT32)
IRSTOREREL $2, [memory], $1

IRSTOREREL $0, [counter], 0
$0 = IRLT $0, THRESHOLD_FIRST_PROFILER
```

it will be necessary to indicate a variable with a specified identifier of value N, the notation `varN` will be used. Finally, the notation `#Label` refers to a generic label.

```
IRBRANCHIF $0, #Label
IRICALL thresholdReached(callSite, method)
IRLABEL #Label
IRVCALL Method, param0, Parameters
```

The only instructions added among the `IRCALL` version enable the profiler to load the *object layout pointer* of the object that actually calls the method (stored ad pointer as the first parameter of the `IRVCALL` instruction – `param0`) and to store it in the correct position inside the available memory; obviously this operation is done only if the object is not null: this check is performed by the `IRBRANCHIFNOT` instruction. The object layout pointer is a value that enables the profiler to distinguish between the different types of objects that could possibly call the virtual method. It is a pointer to structure `ILLayout`[14] and it can be loaded knowing that the first parameter of the virtual call is substantially a variable that contains a pointer to the object body. To read the object layout pointer we have to access to the object header. This can be done if we know the offset position of the variable we are interesting in. Such a offset can be read using the function `getObjectLayoutOffset()` exported by the running garbage collector interface[15].

Both parameters `THRESHOLD_FIRST_PROFILER` and `ZERO_PROFILER_MEMORY_SIZE` assume the values described by the hints read from file or metadata for the callee method, and since they are constant we can directly inject them in the code as constant values.

The main drawback of this approach is that, if this call site is inside a loop, the load and the store instructions for the counter and the branches will be executed the same number of times the callee method will be called. This could lead to performance degradation.

[14] See `iljit/src/layout_manager.[h,c]`.
[15] See file `iljit/src/system_manager.h` and `iljit/src/garbage_collector.[h,c]`.

Second Implementation

To remove the drawback of the previous first profiler implementation we can simply divide the profiler injected code in three blocks.

The first block of code will be placed at method entry and it will simply load the values for the counter and for the threshold reached flag. In such a way they will be loaded just once. This is the code that will be placed at method entry:

```
$0 = IRLOADREL [thresholdReached], 0
$3 = IRLOADREL [counter], 0
```

Now, at call site, the specializer must only place the counter variable increment instruction, as shown in the following piece of code, if the call site is a non-virtual call:

```
$3 = IRADD $3, 1
IRCALL Method
```

If the call site contains a virtual call, we have otherwise to inject also the code to load the type of the objects that calls the method. The result is shown in the following piece of code that has to be placed at call site:

```
IRBRANCHIFNOT param0, #skipTypeStore
$2 = IRLOADREL param0, layoutOffset
$1 = IRREM $0, ZERO_PROFILER_MEMORY_SIZE
$1 = IRMUL $1, sizeof(JITUINT32)
IRSTOREREL $2, [memory] + $1, 0
$3 = IRADD $3, 1
IRLABEL #skipTypeStore
IRVCALL Method, param0, parameters
```

Finally, for each exit point of the caller method, we can store the counter and check if the first profiler threshold has been reached and, eventually, call the native call to inject the method in the recompile request queue. This is the code that must be placed at every method exit point:

```
IRBRANCHIF $0, #Label
IRSTOREREL $1, [counter], 0
$0 = IRLT $1, THRESHOLD_FIRST_PROFILER
IRBRANCHIF $0, #Label
IRICALL thresholdReached(callSite, method)
```

IRLABEL #Label

Anyway, even this way of proceeding can lead to a loss of performance. Suppose that the specializer must deal with the following piece of code:

```
void main(){
    for(i=0;i<200000;i++)
        foo(i, 4);
}

void foo(int a, int b){
    for(int i=0; i < 1000; i++)
        if(i % 400 == 0)
            foo1(a+b,i,0);
        else
            foo2(a+b);
}
```

Consider also that only method `foo1` is specializable. Method `foo` will be called a huge number of times, but method `foo1` only three times for each call of `foo`. If you use the second implementation, anyway, the first profiler counter and the first profiler threshold reached flag will be loaded 200000 times, while only 3000 time would be actually necessary if you use the first implementation model.

For this reason using the second implementation model would be worse than injecting the first one in this case.Therefore, the specializer cannot select, a priori, which is the best implementation model for the zero and first profiler.

5.2.4 Type Dispatching

Once the first profiler has full filled the memory where storing information about the type of the objects that call the virtual method, the specializer will remove the first profiler (i.e. it will remove the instructions from the caller) and will inject the type dispatcher. The structure that describes this dispatcher is called `specializer_method_callSite_type_dispatcher_t` and is shown in listing 5.7. As you can see, this structure only contains the counter used to detect weather this dispatcher should be removed, a threshold reached flag and a pointer to the related call site.

Listing 5.7: The `specializer_method_callSite_type_dispatcher_t` structure

```
typedef struct specializer_method_callSite_type_dispatcher_t{
   void          *callSiteContext;
   JITINT32      counter;
   JITUINT32     thresholdReached;
}
```

First of all it has to extract from the first profiler memory the values of types that have reached the required ZERO_PROFILER_THRESHOLD: this is a very simple operation. After this the profiler can simply inject the dispatcher. An example of a method that has found two constant like types (TYPE_1, TYPE_2) is shown here:

```
$0 = IRLOADREL [counter]

BRANCHIFNOT param0, #skipSpecializedCalls
$1 = IRLOADREL param0, layoutOffset
$2 = IREQ $1,TYPE_1
IRBRANCHIFNOT $2, #skipType1
IRCALL Method_of_type_1, param0, Parameters
$0 = IRSUB $0, 1
IRBRANCH #skipVirtualCall
IRLABEL #skipType1

$1 = IRLOADREL param0, layoutOffset
$2 = IREQ $1,TYPE_2
IRBRANCHIFNOT $2, #skipType2
IRCALL Method_of_type_2, param0, Parameters
$0 = IRSUB $0, 1
IRBRANCH #skipVirtualCall
IRLABEL #skipType2

IRLABEL #skipSpecializedCalls
IRVCALL Method, param0, Parameters
$0 = IRADD $0, 2
IRLABEL #skipVirtualCall

$3 = IRLOADREL [thresholdReached], 0
IRBRANCHIF $3, #skipRecompileCall
IRSTOREREL $0, [counter], 0
$0 = IRLT $0, TYPE_DISPATCHER_REMOVE_THRESHOLD
```

```
IRBRANCHIF $0, #skipRecompileCall
IRICALL thresholdReached(callSite, method)
IRLABEL #skipRecompileCall
```

What the dispatcher does is quite simple. First of all it reads the value of the type dispatcher counter. Next, for each constant like type previously found, it generates an "`if` statement": if the actual type read is equal to the current constant like type, it calls the non-virtual method associated to the object of the given type and decreases the counter value. Otherwise it calls the original virtual method and increment the counter of two unities. Anyway, before load the type of an object that invokes the virtual method it has to check that it is not null. If the value of the counter reaches the threshold and not has been already reached before, then the control flow reaches the native call that will put the method in the recompile request queue.

But how can the specializer find the actual method called (i.e, how can the specializer find a pointer, for example, to method `Method_of_type_1`)? Even this is quite a simple task using the ILJIT framework. Since the specializer has a pointer to the object passed to the method (i.e. the first parameter of the virtual call – param0) and it knows the object layout pointer, it has access to the virtual table of the object: this virtual table is an hash table which returns the ID of actual method that has to be called. Once we have this ID we can retrieve a pointer to the `_ILMethod` structure to use inside the IRCALL[16].

The implementation presented above is not the only one possible for the dispatcher. Another implementation is the following:

```
IRBRANCHIFNOT param0, #skipSpecializedCalls
$1 = IRLOADREL param0, layoutOffset
$2 = IREQ $1,TYPE_1
IRBRANCHIFNOT $2, #skipType1
IRCALL Method_of_type_1, param0, Parameters
IRBRANCH #skipVirtualCall
IRLABEL #skipType1
```

[16]See function `iljit_getMethodFromVirtualMethodAndType` inside the source file `iljit/src/system_manager.[h,c]`.

```
$1 = IRLOADREL param0, layoutOffset
$2 = IREQ $1,TYPE_2
IRBRANCHIFNOT $2, #skipType2
IRCALL Method_of_type_2, param0, Parameters
IRBRANCH #skipVirtualCall
IRLABEL #skipType2

IRLABEL #skipSpecializedCalls
$3 = IRLOADREL [thresholdReached], 0
IRBRANCHIF $3, #skipRecompileCall
$0 = IRLOADREL [counter]
$0 = IRADD $0, 2
IRSTOREREL $0, [counter], 0
$0 = IRLT $0, TYPE_DISPATCHER_REMOVE_THRESHOLD
IRBRANCHIF $0, #skipRecompileCall
IRICALL thresholdReached(callSite, method)
IRLABEL #skipRecompileCall

IRVCALL Method, param0, Parameters

IRLABEL #skipVirtualCall
```

This implementation differs from the previous one because it reduces the number of memory access, since it reads, increments and stores the counter value only if the control flow reaches the virtual call. You should also note that the virtual call will be performed only after the possible native call to the recompile method. This has been done in order to recompile the method as soon as possible, if it is necessary.

Either if at a first look they seems to have the same behaviour this is not true. In the first case the counter can be decremented, so we are supposing that two calls of `IRCALL` compensate one call of the `IRVCALL`. In the second approach, instead, we say that the cost of the `IRVCALL` cannot be compensated, and for this reason the user should set the `TYPE_DISPATCHER_REMOVE_THRESHOLD` at an higher value.

The `TYPE_DISPATCHER_REMOVE_THRESHOLD` is a constant value taken from the hints of called method inside the virtual call.

Listing 5.8: The `specializer_method_callSite_secondProfiler_t` structure

```
typedef struct specializer_method_callSite_secondProfiler_t{
    void          *callSiteContext;
    void          *memory;
    JITUINT32     threshold;
    JITUINT32     thresholdReached;
    JITUINT32     counter;
    JITUINT32     blockSize;
    XanList       *parameters;
} specializer_method_callSite_secondProfiler_t;
```

At this point, anyway, the specializer has not already finished his job: it has to inject the second profiler for the IRCALL injected that are actually specializable. How this is performed will be described in the next section.

5.2.5 Second Profiling

Once the specializer must inject the second profiler (and the possible previously injected first profiler has been removed), it has to initialize the structure for the second profiler itself. The structure `specializer_method_callSite_secondProfiler_t` is shown in listing 5.8. This structure contains a pointer to the memory location where the actual parameters values are stored, the counter used to select where to place the value of a parameter inside the memory, the threshold used to understand if the memory has been full filled, the threshold reached flag and the size of a parameters block (i.e. the parameter K described in section 4.3.3). The list called `parameters` contains which are the actual parameters that can be specialized; each element of this list is of type `specializer_callSite_secondProfiler_param_t` (see listing 5.9). All these values for the second profiler are initialized as described in 4.3.3.

Now that we know which parameters are specializable and how, we can inject the second profiler in the caller method. Suppose that the call has three parameters, but only the first and the third are specializable, and consider also that the

Listing 5.9: The `specializer_callSite_secondProfiler_param_t` structure

```
typedef struct specializer_callSite_secondProfiler_param_t{
   ir_item_t     item;
   JITUINT32     position;
   JITUINT32     offset;
   JITBOOLEAN    _isVector;
} specializer_callSite_secondProfiler_param_t;
```

first parameter is a scalar parameter of type integer, while the third is a vector parameter, so for this last parameter we want to store all the possible length that the vector can assume, if the vector is not null obviously. This is the code that can be injected:

```
$0 = IRLOADREL [thresholdReached], 0
IRBRANCHIF $0, #skipSpecialization

$0 = IRLOADREL [counter], 0
$1 = IRMUL $0, blockSize

$2 = IRADD $1, [memory] + param0->offset
IRSTOREREL param0, [$2], 0

IRBRANCHIFNOT param2, #skipVectorStore
$3 = IRLOADREL param2, arrayLengthOffset
$2 = IRADD $1, [memory] + param2->offset
IRSTOREREL $3, [$2], 0
IRLABEL #skipVectorStore

$0 = IRADD $0, 1
$4 = IRLT $0, threshold
IRBRANCHIF $4, #skipRecompileCall
IRICALL thresholdReached(callSite, method)
IRLABEL #skipRecompileCall

$0 = IRREM $0, threshold
IRSTOREREL $0, [counter], 0
IRLABEL #skipSpecialization
IRCALL Method, param0, param1, param2
```

First of all the second profiler checks if the threshold has not been already reached, loading the threshold reached flag. Next it loads the counter and detects in which

block the profiler has to place the actual value. After this, it loads the required value for the parameters and stores them in the correct memory position using the offset value associated at each parameter. Finally it increments the counter and checks if it exceeds the threshold. If so, the `thresholdReached` method will be called and the caller method will be recompiled in order to remove the second profiler and, if possible, inject the dispatcher or the first profiler again.

To read the length of the vector passed as `param2` we have used as offset the value `arrayLengthOffset`. Also this value can be found using the garbage collector interface as for the `layoutOffset` described above.

Also this dispatcher can be divided as the first profiler in three parts to possibly obtain better results.

Constant Like Expressions

Before generating the constant like expression, for each specializable call parameter the specializer has to compute the values actually assumed by specializable parameters and to calculate their frequency. To store this kind of information we have to use the structure `specializer_multiValueContainer` and `specializer_valueWithFrequency_t`, defined respectively in listing 5.10 and 5.11.

The first structure has the aim to serve as container for values that could theoretically assume different types. The second structure, instead, will be used to associate to a value – with an unknown a priori type – a frequency. With these two structures we can associate to a given parameter a list of possible values, each with its own frequency value. To store this information the specializer uses the structure `specializer_callSite_secondProfiler_param_values_t` described in listing 5.12. To find the constant like values for each parameter the algorithm described in sections 4.3.3 and 4.4.1 has been used.

To describe a generic constant like expression the structure `specializer_staticExpression_hint_t` shown in listing 5.13 has been used.

Listing 5.10: The `specializer_multiValueContainer` structure

```
typedef struct specializer_multiValueContainer{
   union{
        JITNUINT        nuint;
        JITUINT8        uint8;
        JITUINT16       uint16;
        JITUINT32       uint32;
        JITUINT64       uint64;
        JITNINT         nint;
        JITINT8         int8;
        JITINT16        int16;
        JITINT32        int32;
        JITINT64        int64;
        JITNFLOAT       nfloat;
        JITFLOAT32      float32;
        JITFLOAT64      float64;
   } value;
   JITUINT32 type;
}specializer_multiValueContainer;
```

Listing 5.11: The `specializer_valueWithFrequency_t` structure

```
typedef struct specializer_valueWithFrequency_t{
   specializer_multiValueContainer valueContainer;
   JITFLOAT32    frequency;
} specializer_valueWithFrequency_t;
```

Listing 5.12: The `specializer_callSite_secondProfiler_param_values_t` structure

```
typedef struct
        specializer_callSite_secondProfiler_param_values_t{
   JITUINT32 position;
   XanList* values;
} specializer_callSite_secondProfiler_param_values_t;
```

Listing 5.13: The `specializer_staticExpression_hint_t` structure

```
typedef struct specializer_staticExpression_hint_t{
   specializer_expression_hint_t          *expression;
   specializer_multiValueContainer        *values;
} specializer_staticExpression_hint_t;
```

Listing 5.14: The `specializer_method_callSite_dispatcher_t` structure

```
typedef struct specializer_method_callSite_dispatcher_t{
   void            *callSiteContext;
   JITINT32        counter;
   JITUINT32       thresholdReached;
} specializer_method_callSite_dispatcher_t;
```

This structure just associate at each sub-expression of the `expression` field a value contained in the values vector. Since at this point we know that the expression is constant like, the specializer can safely ignore the frequency of each values in the `values` vector.

5.2.6 Dispatching

After the specializer has removed the second profiler from the caller method, if there is at least one constant like expression, it can proceed in the injection of the dispatcher. The dispatcher is described by the structure `specializer_method_callSite_dispatcher_t` shown in listing 5.14. As you can see this structure only contains the counter, the threshold reached flag and the related call site pointer.

To better understand how the dispatcher is injected, consider the same method described in section 5.2.5, and suppose that the specialer knows the following two constant like expressions:

```
(1) #1 STATIC AND #3 STATIC LENGTH
                Values : [#1 = 1, #3 = 30]

(2) #1 EQUALS 0
                Values : [#1 = 0]
```

For these constant like expressions we obtain the following dispatcher:

```
$0 = IRLOADREL [counter]

$1 = IREQ param0, 1
IRBRANCHIFNOT param2, #skipSpecialized1
$3 = IRLOADREL param2, arrayLengthOffset
$2 = IREQ $3, 30
$1 = IRAND $1, $2
IRBRANCHIFNOT $1, #skipSpecialized1
IRCALL SpecializedMethod2, param1
$0 = IRSUB $0, 1
IRBRANCH #skipOriginalCall
IRLABEL #skipSpecialized1

$1 = IREQ param0, 0
IRBRANCHIFNOT $1, #skipSpecialized2
IRCALL SpecializedMethod2, param1, param2
$0 = IRSUB $0, 1
IRBRANCH #skipOriginalCall
IRLABEL #skipSpecialized2

IRCALL Method, param0, param1, param2
$0 = IRADD $0, 2
IRLABEL #skipOriginalCall

$4 = IRLOADREL [thresholdReached], 0
IRBRANCHIF $4, #skipRecompileCall
IRSTOREREL $0, [counter], 0
$0 = IRLT $0, DISPATCHER_REMOVE_THRESHOLD
IRBRANCHIF $0, #skipRecompileCall
IRICALL thresholdReached(callSite, method)
IRLABEL #skipRecompileCall
```

First of all the dispatcher loads the counter value. Next, for each constant like expression previously found,it generates an "`if` statement" that check if the conditions to call the specialized method applies, taking care to do not call the specialized method if the vector is null. If this does not happen, the original method will be called. Finally the counter will be stored in memory with his new value, and, if its value will exceed the dispatcher threshold (that has been taken from the hints of the callee method read from file) the method will be recompiled and the dispatcher removed. The dispatcher removal process can be performed

simply by removing the instructions injected. Anyway, the counter store instruction and the indirect call instruction will not take place if the dispatcher remove threshold has already been reached.

In this approach the counter is decremented if the control flow reaches a specialized call, while it will be incremented of two unities if the control flow reaches the original call. Moreover all the call site generated at this point will be marked as `UN-SPECIALIZABLE`.

Just like the type dispatcher, we can manipulate this one in order to obtain a different version that access the memory in order to read the counter only when the control flow reaches the original call:

```
$1 = IREQ param0, 1
IRBRANCHIFNOT param2, #skipSpecialized1
$3 = IRLOADREL param2, arrayLengthOffset
$2 = IREQ $3, 30
$1 = IRAND $1, $2
IRBRANCHIFNOT $1, #skipSpecialized1
IRCALL SpecializedMethod2, param1
IRBRANCH #skipOriginalCall
IRLABEL #skipSpecialized1

$1 = IREQ param0, 0
IRBRANCHIFNOT $1, #skipSpecialized2
IRCALL SpecializedMethod2, param1, param2
IRBRANCH #skipOriginalCall
IRLABEL #skipSpecialized2

$2 = IRLOADREL [thresholdReahced], 0
IRBRANCHIF $2, #skipRecompileCall
$0 = IRLOADREL [counter], 0
$0 = IRADD $0, 1
IRSTOREREL $0, [counter], 0
$0 = IRLT $0, DISPATCHER_REMOVE_THRESHOLD
IRBRANCHIF $0, #skipRecompileCall
IRICALL thresholdReached(callSite, method)
IRLABEL #skipRecompileCall

IRCALL Method, param0, param1, param2

IRLABEL #skipOriginalCall
```

The same discussion done for the two different kinds of type dispatcher in section 5.2.4 can be applied also in this context.

Callee Method Specialization

One thing that we have not considered previously in this section, is how the specializer generates a specialized version of a given IR method. This process [17] can be done in four steps: *Signature creation*, *Method cloning*, *Constant values injection*, *Variables Shift*.

Signature Creation In this phase the specializer creates CIL, IR and JIT (i.e. machine) signature. To perform this operation the specializer has only to copy the CIL signature of the original method skipping the parameters that have been specialized and that are not vector parameters. Next, it will automatically create the IR signature and the JIT signature. They can be derived from the CIL signature using respectively the functions `decode_ir_signature` and `decode_jit_signature` provided by the ILDJIT framework[18]. For example, suppose that the CIL signature of the original method is "`System.Int32 foo(System.UInt32, System.Int32[], System.Float32)`" and you have a constant like expression that says that you have to specialize the first and the second parameter. The specialized signature will be "`System.Int32 foo(System.Int32[], System.Float32)`".

Method cloning This phase is quite trivial. The specializer only has to clone each method instruction as it is, without any modification. In this way the specializer obtains a perfect copy of the original method that it can modify in the subsequent phases.

[17] See the function `specializer_clone_makeSpecializedIRMethod` in `libirprofiler/src/irspecializer_util.[h,c]`

[18] See files `iljit/src/cil_ir_translator.[h,c]`

Constant values injection In this phase the specialer injects inside the generated method the store instructions that substitute the removed parameters. For example, consider the first expression described previously in this chapter (`#1 STATIC AND #2 STATIC LENGTH; Values : [#1=1,#2=30]`) and consider also that, in the ILDJIT compiler, the first n^{th} variables refers to the n^{th} parameters. For this reason, inside the generated method, we have to inject something like this:

```
IRSTORE var0 , 1
```

because `var0` refers to the first parameter which in the expression has been identificated with position 1.

What it has been said since here can only be applied to non-vector parameters. For this kind of parameters the specializer has to find all the aliases of the variable that identify the vector. Next, the specializer has to find all the `IRLOADREL` instructions that have as source the variable itself (or one of its aliases) and that have as offset the value of `arrayLengthOffset`. For example, consider the following piece of code:

```
var5 = IRLOADREL var2 , 2
var6 = IRLOADREL var2 , arrayLengthOffset
IRSTORE var7 , var2
var8 = IRLOADREL var7 , arrayLengthOffset
var10 = IRLOADREL var9 , arrayLengthOffset
```

Suppose now that only `var7` is an alias for `var2`. The code will be translated as follows:

```
var5 = IRLOADREL var2 , 2
IRSTORE var6 , 30
IRSTORE var7 , var2
IRSTORE var8 , 30
var10 = IRLOADREL var9 , arrayLengthOffset
```

In this way the specializer replaces every `IRLOADREL` that attempts to read the length of the vector with a more economic `IRSTORE`.

Variables Shift As already mentioned, the first n^{th} variables refers to the n^{th} parameters, for this reason `var0` refers to the first parameter, `var1` refers to the second parameter and so on. Since we have changed the signature of the specialized method this could not be true: the specializer must shift the value of the identifier of the variables in order to maintain this notation.

First of all the specializer, to perform this task, generates a *substitute vector*: this vector contains, for each parameter that has been removed from the signature, the new value of the identifier of variable. For example, considering again the constant like expression `#1 STATIC AND #2 STATIC LENGTH; Values : [#1=1, #2=30]`. We have to remove only the first element in the signature; for this reason the substitute vector will contain only a value for the first parameter: this value will be used as the brand new ID for `var0` inside the code. Consequently all others identifier for the variable must be shifted if it is necessary.

For example, suppose that the specializer finds the following piece of code that must be variable-shifted:

```
var4 = IRADD var0, var7
var4 = IRADD var3, var0
var5 = IRLOADREL var2, 2
IRSTORE var6, 30
IRSTORE var7, var2
IRSTORE var8, 30
var10 = IRLOADREL var9, arrayLengthOffset
```

Now suppose that the new ID for `var0` will be `var99`. The new code, after the ID variable shift, will be:

```
var3 = IRADD var99, var6
var2 = IRADD var2, var99
var4 = IRLOADREL var1, 2
IRSTORE var5, 30
IRSTORE var6, var2
IRSTORE var7, 30
var9 = IRLOADREL var8, arrayLengthOffset
```

As you can see, all the variable with ID over the zero one, have been decremented by one unity. The same approach can be applied for any number of

parameters that must be removed from the signature.

5.3 Multi-threading program issues

The specializer description that has been done since here has never kept into account the fact that a program is not always composed by a single thread. In a multi-threading environment the same method can be executed by two different thread with two possibly complete different behaviour, depending just on the input parameters of the threads.

In the case of a multi-threading environment there are actually two different kind of problems. The first one is a classical mutual exclusion problem between the memory location used by the profilers or by the dispatcher to store information about the state of the call site (for example the counter variable of the first profiler). Suppose that two thread access to the same profiler call site: if a

The second kind of problem concern the specialization itself. As already mentioned two thread with the same code can possibly have two completely different behaviour: while one can be worthwhile specializing, the second would not. Anyway, since both threads share the same profiler information they interfere each other leading to a possibly bad run-time performance.

One way to solve this problem would be to associate a specialization structure at each thread. In this way each thread would have its own profiling information at the cost of a very high data and code duplication, because the specializer should also duplicate the code of the method where the call site resides because a call site could be in a precise state in a thread but in a completely different state in another one.

Another way to solve this problem would be to put in the injected code of the profilers and the dispatcher some instructions in order to detect which is the thread that is executing the code. While this approach could be easier than the previous one, it could anyway lead to a very huge performance degradation.

For all these reasons the multi-threading problem has not been take into account while developing the described approach to the specialization process.

5.4 Final Implementation Notes

What has been said in this chapter broadly describes how the system has been implemented. Anyway a number of technical details has been skipped in order to keep the description as simple as possible. For example, when a dispatcher must be injected the specializer always check for some particular case:

- The injected call will always be called: this could happen when each actual parameter of the specialized method is a constant value;
- A method call has already been injected in the same dispatcher: this could happen when two constant like expression are redundant;

Other optimizations has been used for the profiler implementation. For example, before injecting the second profiler the specializer verifies if every parameter of the method is constant: if so the second profiler will no be injected, but the dispatcher will be.

Obviously, for the method specialization, others implementation techniques were possible. For example, instead of generating every time a method it could be worthwhile (but a little more complicated) using the *slicing* approach, as described in [40].

Chapter 6

Experimental Results

In this chapter the results of the implementation of the specialization procedure inside ILDJIT dynamic compiler will be presented[1]. Numeric results were been obtained running the compiler with three optimization levels:

Level 0 : No optimization will be performed at all;

Level 3 : Only standard optimizations will be performed;

Level 5 : The compiler will execute the standard optimizations and the specializing procedure on the code.

These executions provides information about the running time of the benchmarks.

For each benchmarks used will be shown a list of possible parameters that can influence the program execution. We will try to modify to these parameters to detect when the specialization procedure actually leads to benefits. The experiments has been execute on Linux on top of Intel Pentium hardware.

6.1 Operating Mode

In order to determinate the speed-up (or the eventual slow-down) gained by the use of the specialization procedure, the specializer must understand which actual methods can be specialized and how. As already mentioned this can be done by

[1]The 0.0.3.1 version of the ILDJIT compiler has been used

an external tool or compiler but, since nothing has already been developed, in order to find this information we have to follow a different approach.

This approach involves the activation of the *full specialization mode* inside the specializer. This operating mode tries to specialize everything if possible. This can be simply done by generating, for each method, a new specialization hint. Every time the specializer founds a method which has no specialization hints it follows these two steps:

- Generate a new specialization hints for the methods with default parameters (i.e. profiler and dispatchers thresholds and memory sizes)[2];
- For each parameter that can be specialized (i.e. for each scalar or vector parameter, excluding in this way pointers and objects) will be generated an expression hint which action will be constant like.

Now, We can run the required program in this mode many times, increasing the specialization hints parameters. In this way we can manually understand which method are worthwhile specializing.

In appendix A, for each benchmark that will be described in section 6.2 you will find which hints have been found for for methods.

Obviously this approach is far from generating optimal hint expressions for the program methods, since it simply detects if a method parameter is constant like and does not perform a costs and benefits analysis. For example, thought the specializer knows that a parameter is constant like, this information could not influence the specialization of the method because this parameter is not used at all, or the method to specialize is empty.

For all these reasons the results obtained from the full specialization mode can be influenced by the problem described.

[2]These parameters can be changed modifying the file "specializer.spc". This file contains a multiply factor that is used to modify the default values in full specialization mode. See files `libiljitirprofiler/src/irspecializer.[h,c]` for more details.

6.2 Benchmarks description

The benchmarks that will be employed are the following:

- a classic *BucketSort* algorithm (`bucketsort`). This program sorts vectors which size ranges between two given values for a given number of times;
- a classic *BubbleSort* algorithm (`bubblesort`). This program sorts vectors which size ranges between two given values for a given number of times;
- a *DES* cryptographic algorithm (`des`). This program apply the DES algorithm on a given set of different byte vectors for a given number of times;
- a *context-adaptive variable-length decoding* algorithm provided by Heinrich Hertz Institute (`decoder`);
- a *spectral normal matrix* numeric computation algorithm (`spectarlnormal`). This program approximate the spectral normal of a matrix that has a given size. It repeats the required operation on this matrix a given number of times;
- a classic *sieve of Eratosthenes* algorithm (`sieve`). This program applies the sieve algorithm on vectors which size range between two given values;
- a classic *prime number identification* algorithm (`testprime`). This program applies the algorithm on vectors of integers numbers. The size of the vectors range between given values. Also the numbers that has to be tested inside the vectors lies in a given range of numbers;
- a classic non-recursive *factorial* algorithm (`factorial`) that for numbers greater than 80 uses the Stirling approximation. The operation is performed a given number of times;

- a classic non-recursive *fibonacci* algorithm (`fibonacci`) that for numbers grater than 80 uses the Binet's formula. The operation is performed a given number of times.

As you can see, these programs represents a spectrum of different application fields: video decoding, cryptography, sorting and numerical algorithm. All these program are taken from other benchmark suites (see [31, 32, 41, 42, 43]) and slightly modified, where possible, in order to generate programs that can be parametrized from the command line, without compromising the main algorithm execution. Table 6.2 summarize, for each program, these parameters.

Table 6.1: Benchmark program and parameters

Program	**Language**	**Parameters**	**Description**
`bucketsort`	`C#`	`E`, `I`, `R`, `S`	Executes `E` times the method that calls the `bucketsort` procedure. The method that calls the algorithm execute it `I` times. The vectors to sort have size between `S` and `(S+R-1)`.
`bubblesort`	`C#`	`E`, `I`, `R`, `S`	Same parameters as bucketsort.
`des`	`C#`	`E`, `I`	Executes `E` times the method that calls the `des` procedure. The method that calls the algorithm executes it `I` times.
`decoder`	`C`	None	–
`spectarlnormal`	`C#`	N, C	Executes `C` times the main approximation algorithm on a matrix which order is `N`.
`sieve`	`C#`	`E`, `I`, `R`, `S`	Same parameters as bucketsort.
`factorial`	`C#`	`E`, `I`, `R`, `S`	Same parameters as bucketsort.
`fibonacci`	`C#`	`E`, `I`, `R`, `S`	Same parameters as bucketsort.

Anyway, due to the high number of parameters that can be used to change the behaviour of the specializer procedure, only some of them for these methods will be considered variable, while other will be taken as constant. For each program that will be analyzed there will be a description of these parameters.

6.3 Experimental results

From the results obtained by the benchmarks we can see that the speedup obtainable from this procedure can vary from 2% to about 38%, which is the classical

range of speed-up values defined in literature [3, 4].

To generate the results presented in appendix B some benchmarks with different values for the variable parameters have been run. Each benchmark has been executed with three different optimization flags:

- **-O0**: no optimization performed.
- **-O3**: standard optimizations and analysis performed (algebraic simplification, basic-block identifier, branch predictor, constant propagation, copy propagation, dead code elimination, escapes detection, induction variables identification, liveness analysis, loop identification, loop invariant code motion, loop invariants identification, null check removal, post-dominators analysis, pre-dominators analysis, reaching definitions analysis, trace scheduler analysis);
- **-O5**: the specialization will be executed and also the standard optimizations will do.

Each benchmark program, with a different combination of optimization flag and input parameters, has been executed 21 times, and for the obtained values the median has been taken[3].

Once obtained the values for each benchmark three different speed up are calculated in order to show how good (or worse) the specialization procedure can be. The computed speed ups are:

- **O3/O0**: the speed up of the optimized version of the program respect the non-optimized one;
- **O5/O0**: the speed up of the specialized version of the program respect the non-optimized one;

[3]The median has been chosen as statistic reference value instead of the mean, because it is not compromised by the outlayers time values inside the data collected. For example, the very first execution of the method is, in most cases, executed in much more time than the other executions due the use of the instruction cache.

- **O5/03**: the speed up of the specializer version of the program respect the optimized-one;

As already mentioned the results of the specialization depend of three main factors: the program it-self, the generated expression hints, the program parameters.

Consider first of all the `bubblesort` benchmark. As you can see from table B the speed up strongly depends on the parameter R (which is the "range" of the vectors length passed to the bubble-sort procedure). For value of $R = 4$ we have negative speed up, while for $R < 4$ we have very good results. This behaviour is caused by the second profiler static threshold as described in table A. The value for this threshold has been fixed to 0.3, this means that if the method parameter assumes more than three constant like values than the specialization should not take place. To allow the specializer to consider constant like much more values we should rise the threshold, but, in this way, the specializer could find much more false constant likes values (see section 4.3.3).

In figures 6.1, 6.2 and 6.3 you can see the speed up results from the execution of the `bubblesort` program with the same value for variable S ($S = 50$). As you can see the higher speed-up[4] can be obtained when $R = 3$ (38%), while for $R = 4$ we can obtain only negative speed up.

The huge speed up that has been obtained can be explained by the fact that the bubble-sort algorithm works on vectors which length is read a lot of times inside the `for` loops placed in the algorithm code. In this case, since the high-level language that has generated the CIL code is `C#`, a lot of extra checks on vector size has been inserted, in order to avoid the user to read or write over the array bounds. Anyway, applying the specialization process to the algorithm all these extra check can be removed, leading to a much faster code. Moreover, again through the specialization process, inside the specialized code any load from

[4]Now with the term speed up will be always intended the speed-up of `O5/O0`.

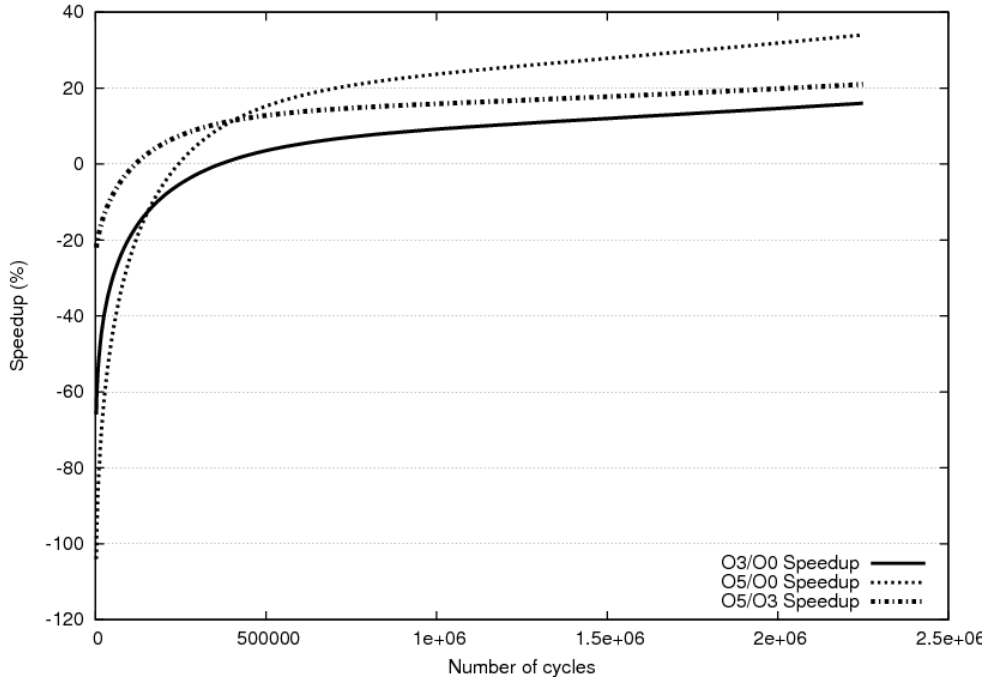

Figure 6.1: `bubblesort` speedup for `S=50` and `R=2`

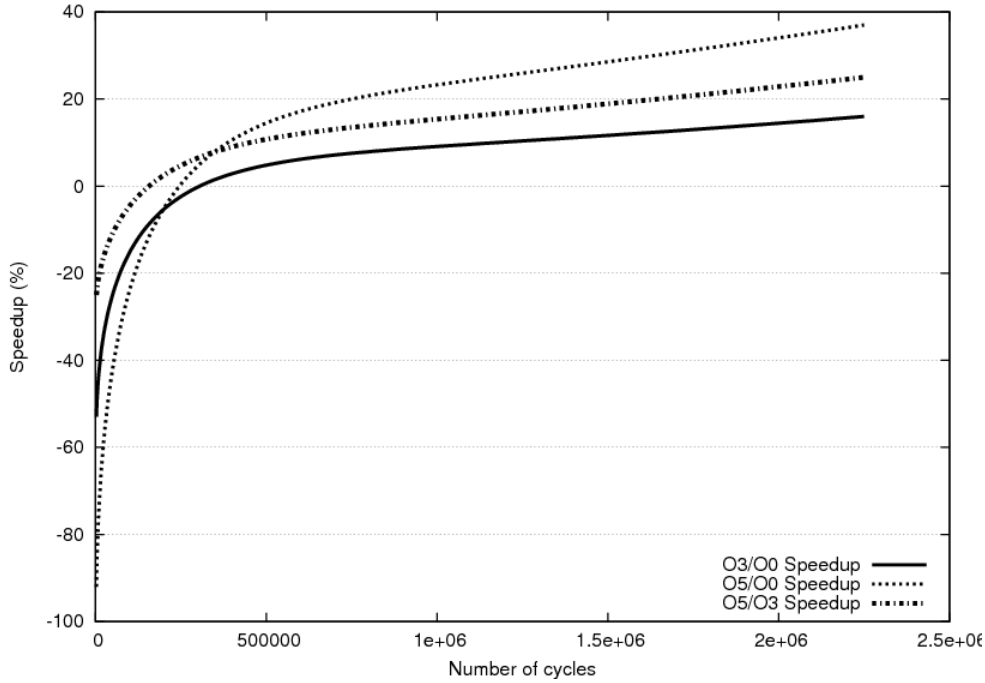

Figure 6.2: `bubblesort` speedup for `S=50` and `R=3`

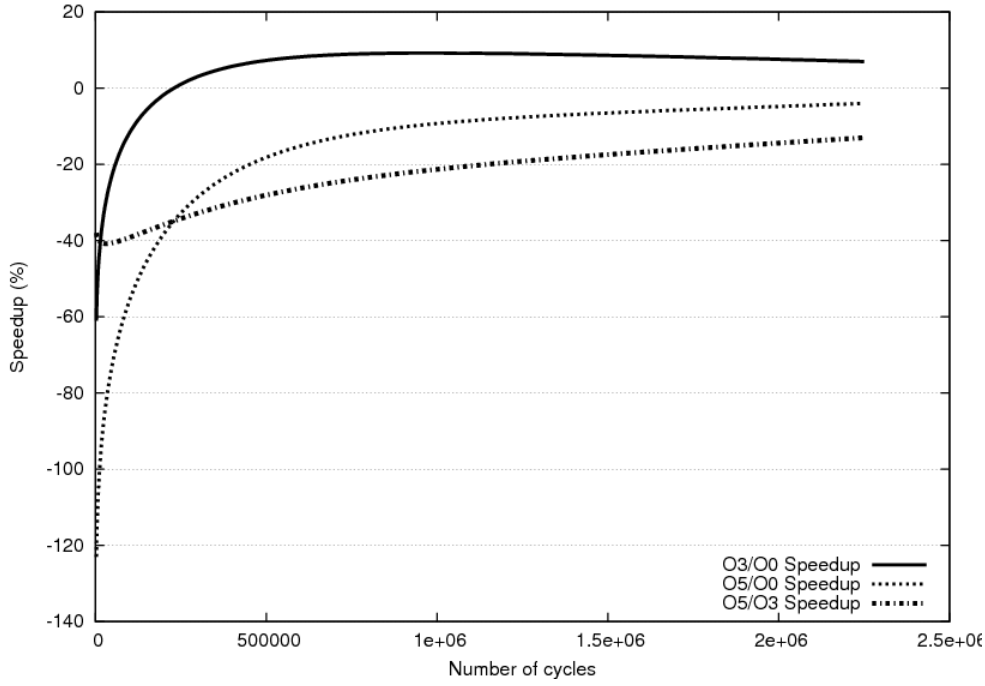

Figure 6.3: `bubblesort` speedup for `S=50` and `R=4`

memory of the vector size has been replaced by a constant value: this has dramatically reduced the time spend inside the main specialized bubblesort function that actually performs the algorithm.

The same reasoning can be applied to the `bucketsort` benchmark. Also here the speed-up result, that are shown in table B, are strongly dependent on he R parameter and on the second profiler threshold that has been fixed to 0.3 as described in table A: for value of $R = 4$ we have negative speed up, while for $R < 4$ we have very good results.

In figures 6.4, 6.5 and 6.6 you can see the speed up results from the execution of the `bucketsort` program with the same value for variable S ($S = 50$). Even here we can explain the speed up obtained by the specialization process with the boundary check removal performed on the vectors manipulated by the algorithm. Anyway, the speed up obtain is lower than the one obtained by the `bubblesort` benchmark due the different kind of sorting algorithm: while in the bubblesort

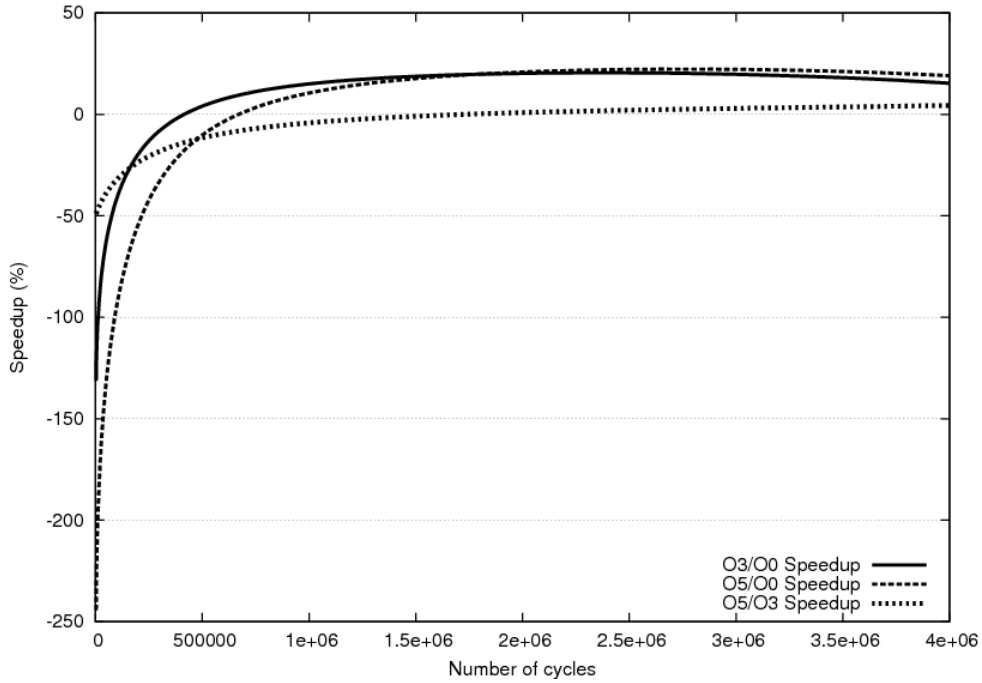

Figure 6.4: `bucketsort` speedup for `S=50` and `R=2`

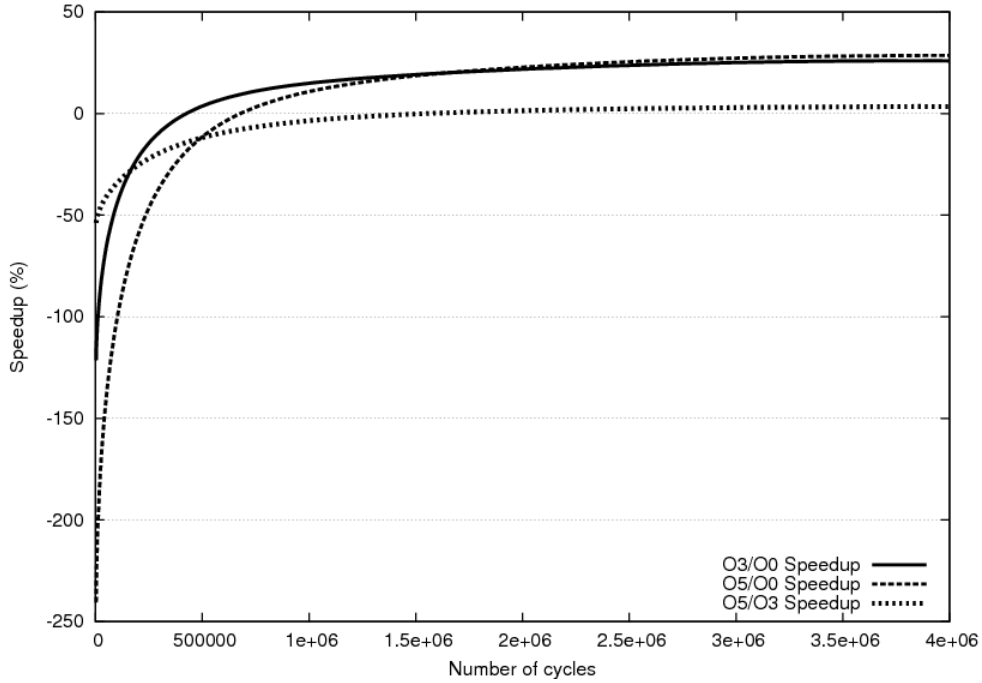

Figure 6.5: `bucketsort` speedup for `S=50` and `R=3`

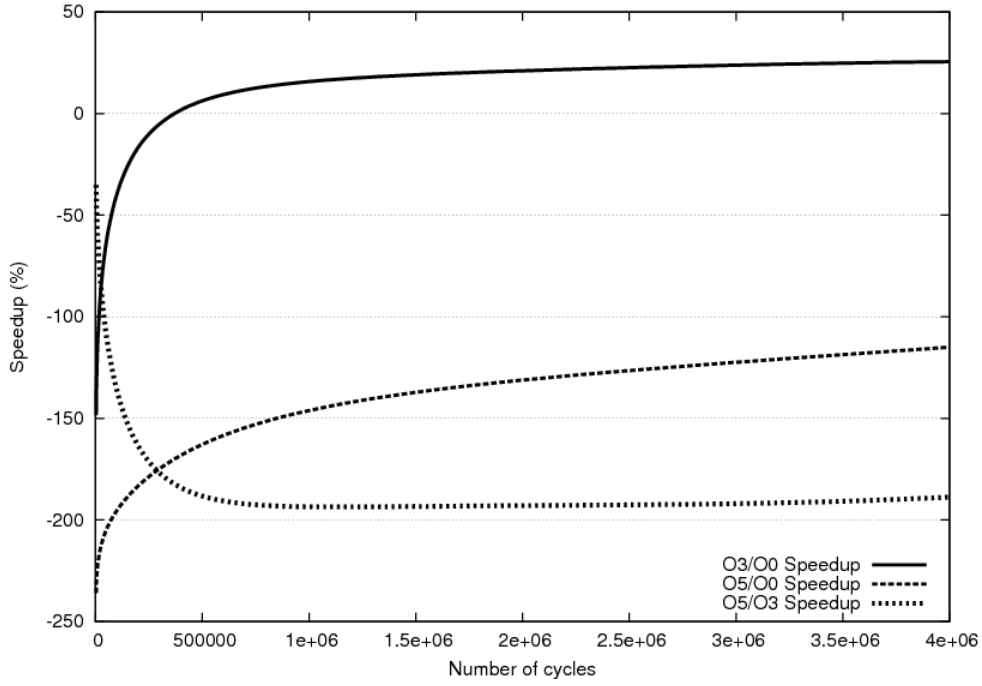

Figure 6.6: `bucketsort` speedup for `S=50` and `R=4`

algorithm two nested `for` loops are present, in the implementation used by the bucket algorithm there is only one `for` loop. Also for the bucketsort algorithm a big contribution to the speed up has been derived by the fact that any load from memory of the vector size has been replaced by a much more cheap constant value assignment that does not involves the use of the memory.

Not only the sorting algorithms presented above work on vectors, also the `des` algorithm manipulate vectors of byte with different sizes. The result of the specialization are placed in tables B. As you can see in the specialization parameters in table A a lot of methods can be specialized. Each method works on a vector with a different size.

In figure 6.7 you can see the speed-up obtained by the execution of the `des` algorithm with a different number of cycles. The main explanation of why the specializer gives a speed-up of about $5\% \div 6\%$ is because also this algorithm works on vectors and a lot of bound checks can be safely removed inside the specialized

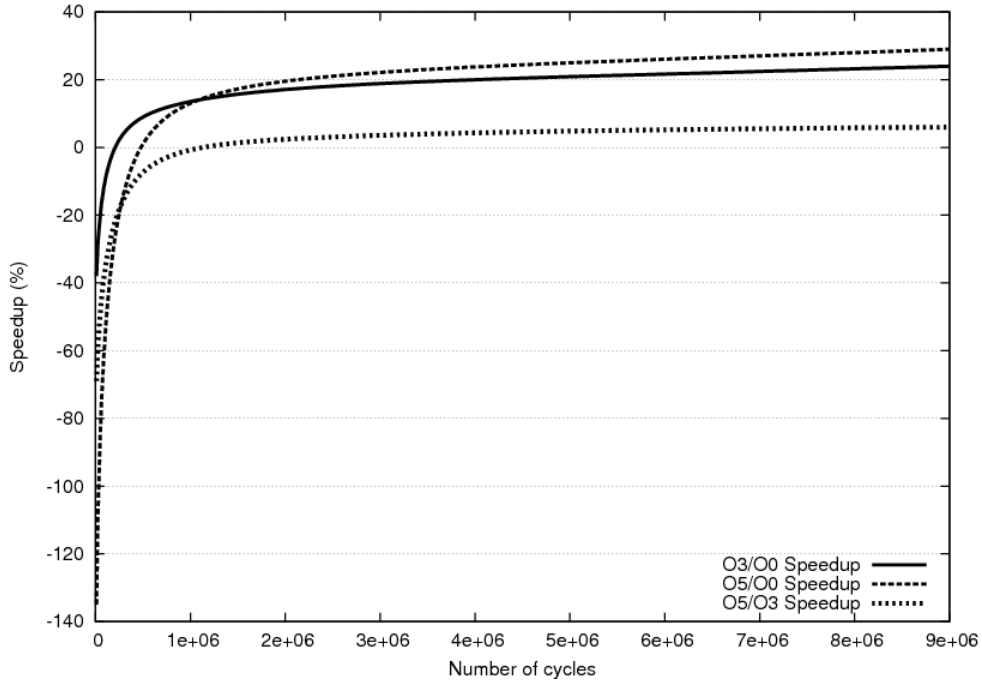

Figure 6.7: `des` speedup

code. Moreover, also any load from memory of the vector size has been avoided since we already know the vector size, through the specialization process.

The `sieve` algorithm works on vectors too, so we can expect about the same speed up obtained in the `des` benchmark. In fact, as you can see in table B, using the specialization hints described in table A you can again obtain a speed up that ranges between $5\% \div 6\%$.

In figures 6.8, 6.9 and 6.10 you can see the speed up results obtained from the execution of the `sieve` benchmark with the same value for variable S ($S = 1000$). The explanation of such a speed up can be again derived by the fact that sieve algorithm implemented makes use of vectors and, with specialization, bound checks and load from memory of the vector size can be avoided.

The last benchmark that works on vectors is the `spectralnorm` program. The hints used to produce the result are shown in table A, while the numeric results are presented in table B.

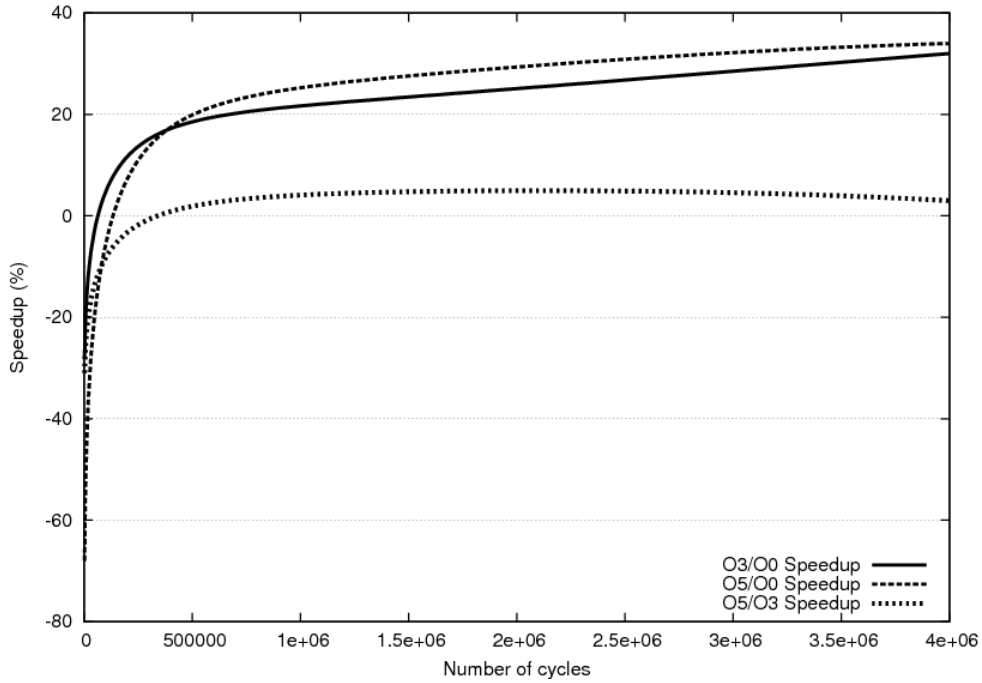

Figure 6.8: `sieve` speedup for `S=1000` and `R=2`

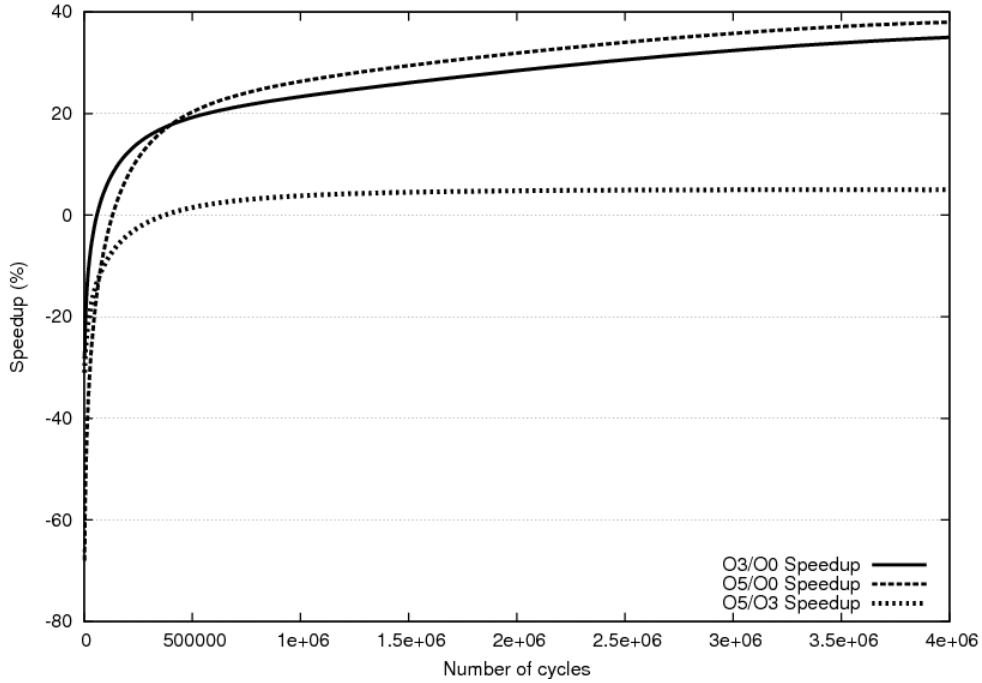

Figure 6.9: `sieve` speedup for `S=1000` and `R=3`

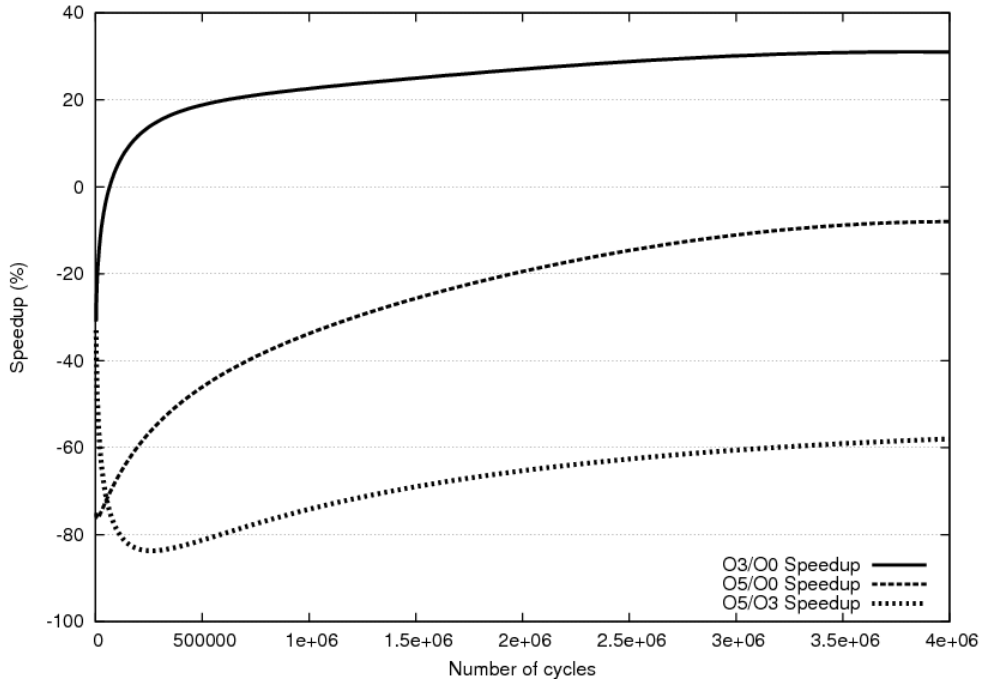

Figure 6.10: `sieve` speedup for `S=1000` and `R=4`

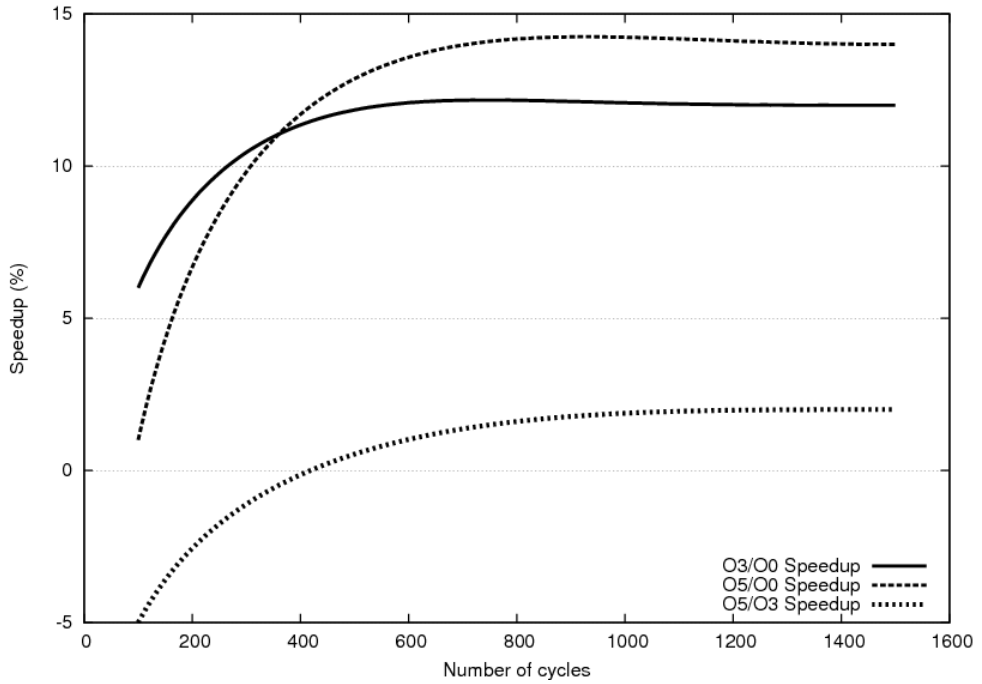

Figure 6.11: `spectralnorm` speedup for `N=300`

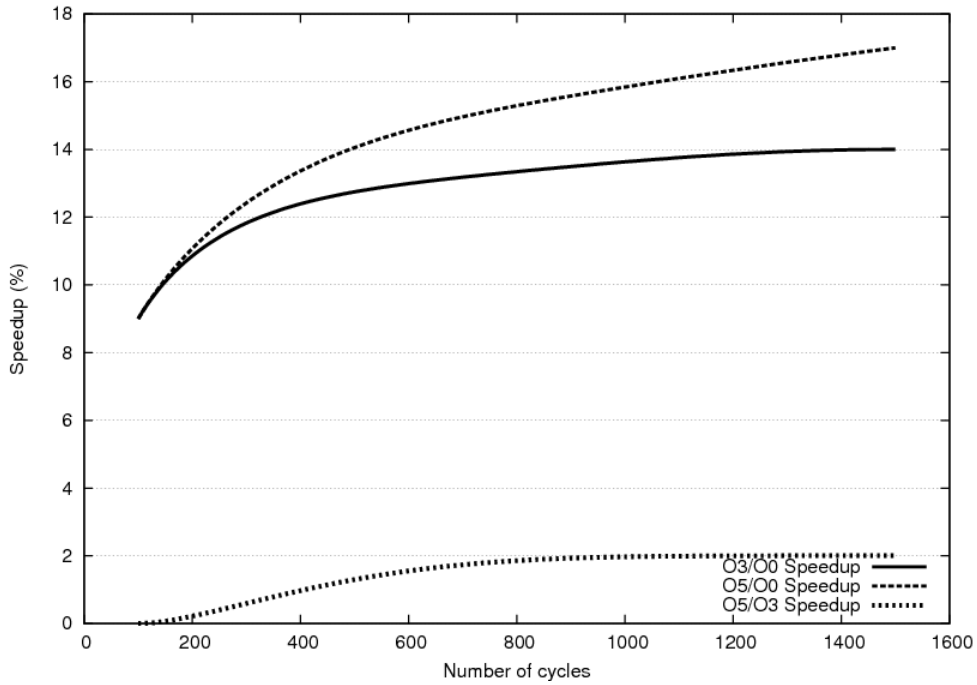

Figure 6.12: `spectralnorm` speedup for `N=400`

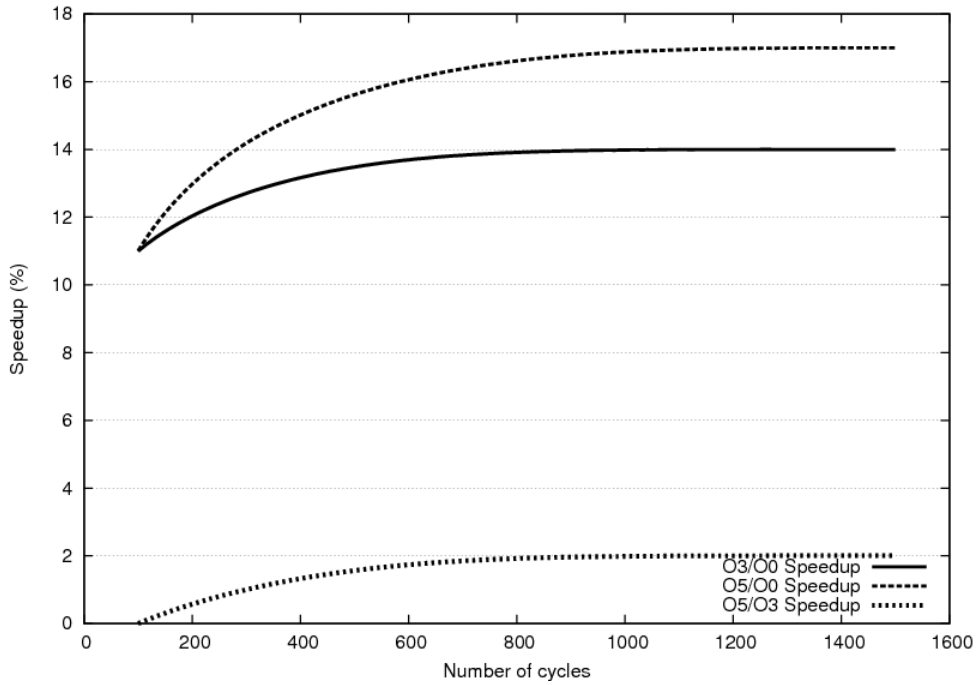

Figure 6.13: `spectralnorm` speedup for `N=500`

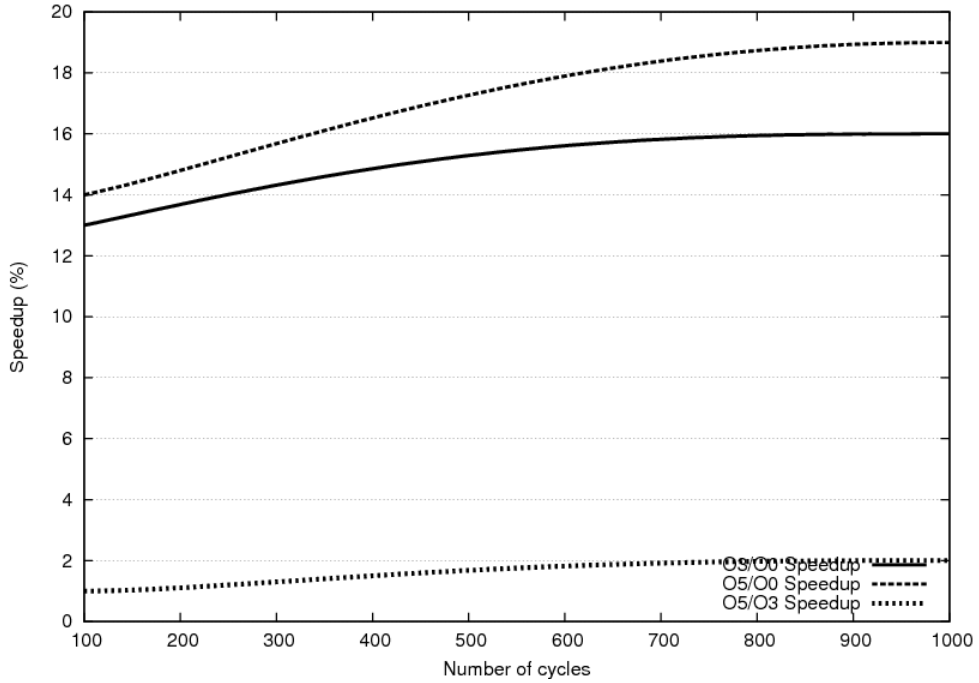

Figure 6.14: `spectralnorm` speedup for `N=600`

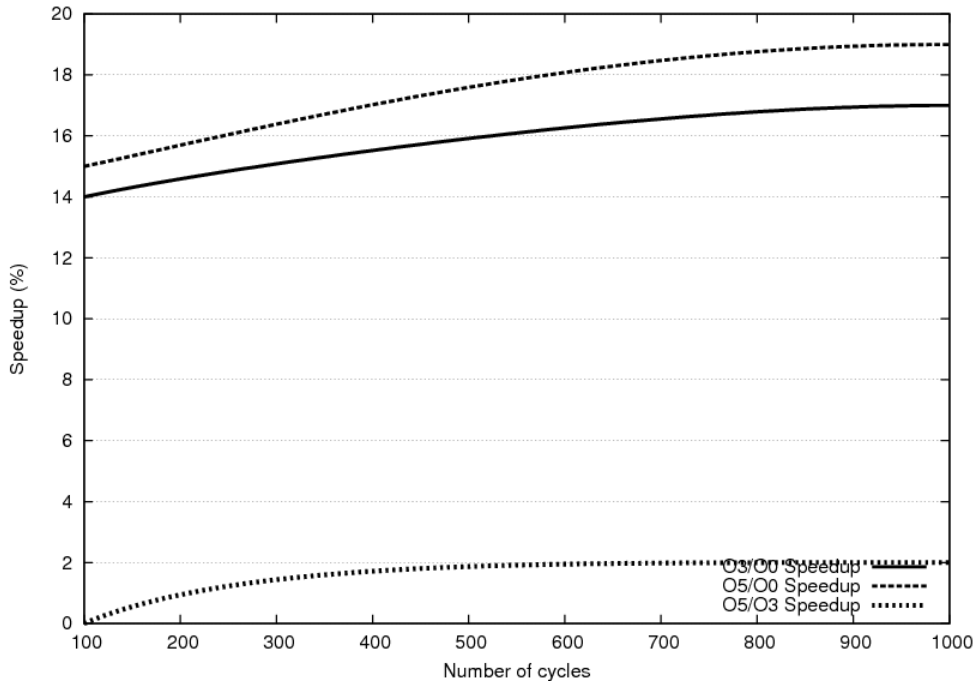

Figure 6.15: `spectralnorm` speedup for `N=700`

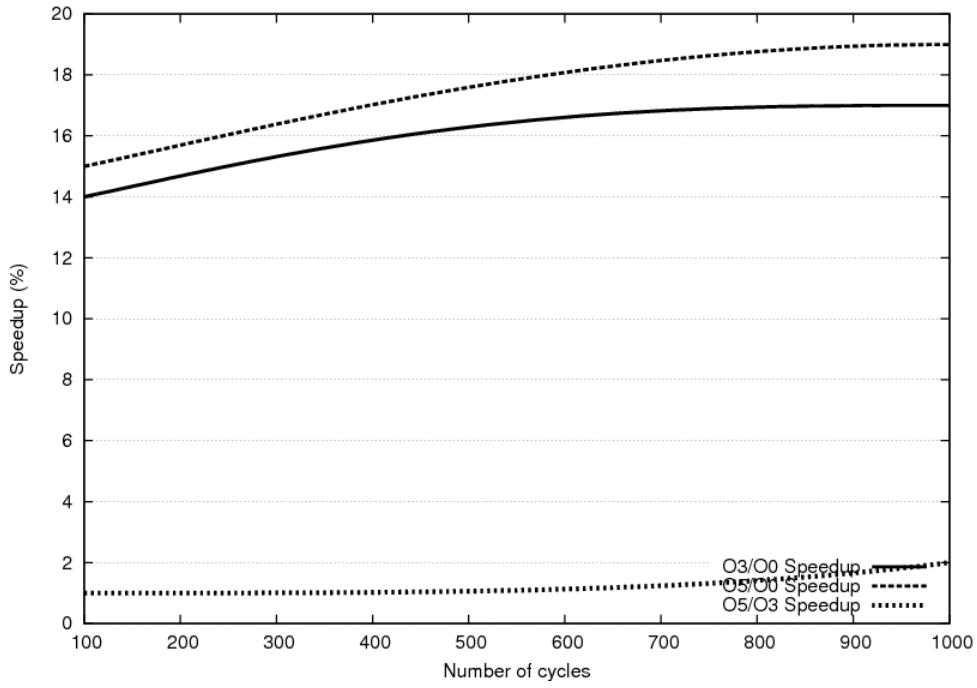

Figure 6.16: `spectralnorm` speedup for `N=800`

The figures 6.11, 6.12, 6.13, 6.14, 6.15 and 6.16 show the speed up obtained from the execution of the spectralnorm benchmark on different matrix sizes. As you can see the speedup gained by the use of specialization is about of $2\% \div 3\%$. Even here the main advantages of the specialization lays either in the bound check removal on the array length and in the removal of any load from memory that perform the read of the memory size.

Now two benchmarks that does not work on vectors will be presented.

Consider first of all the `factorial` benchmark. You can find numeric results in table B, while the specialization hints are reported in table A.

As you can see in figures 6.17, 6.18, 6.19 and 6.20, in the best case the specialization process lead to a speed up of about $2\% \div 3\%$, that it is smaller that what has been obtained with the benchmarks that uses vectors. This speed up can be explained by the fact that in the `fibonacci` algorithm the specializer trays to perform specialization substituting some parameters with the actual value, so what

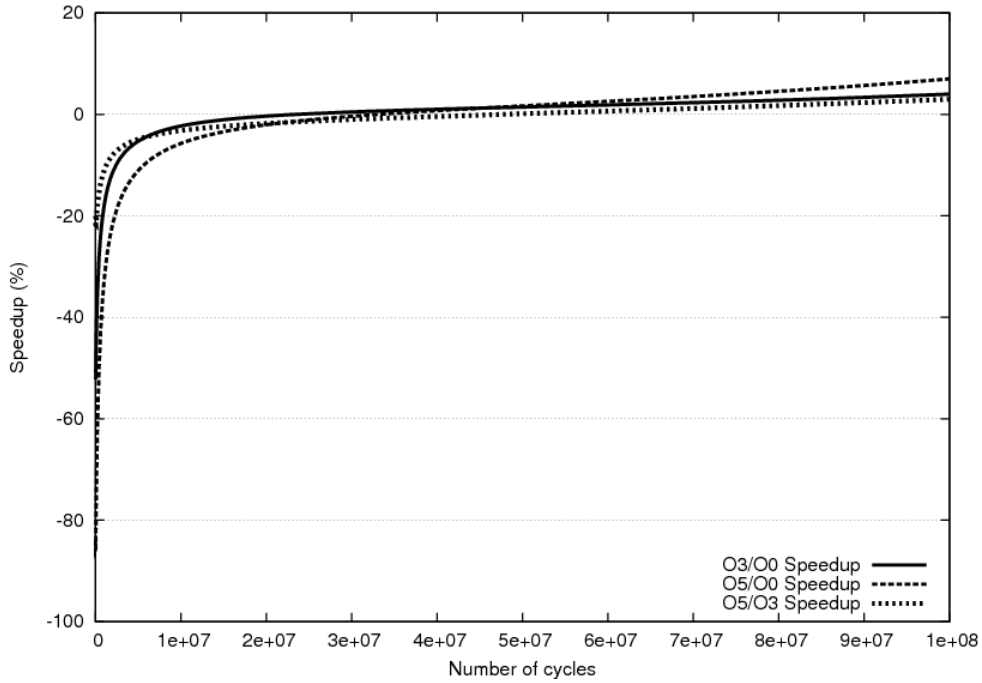

Figure 6.17: `factorial` speedup for `S=79` and `R=2`

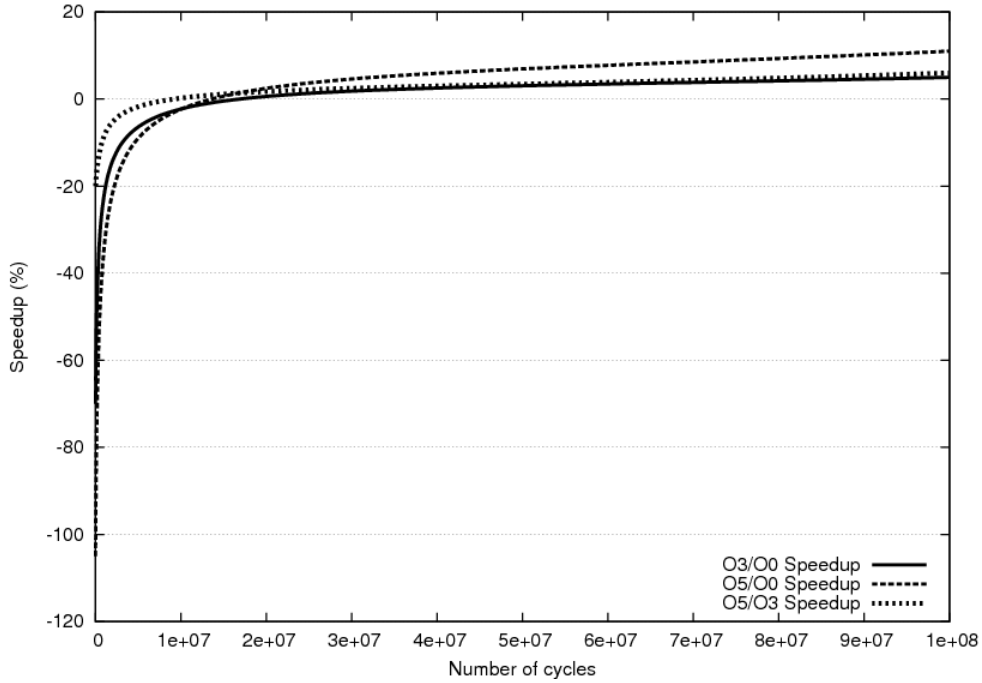

Figure 6.18: `factorial` speedup for `S=79` and `R=3`

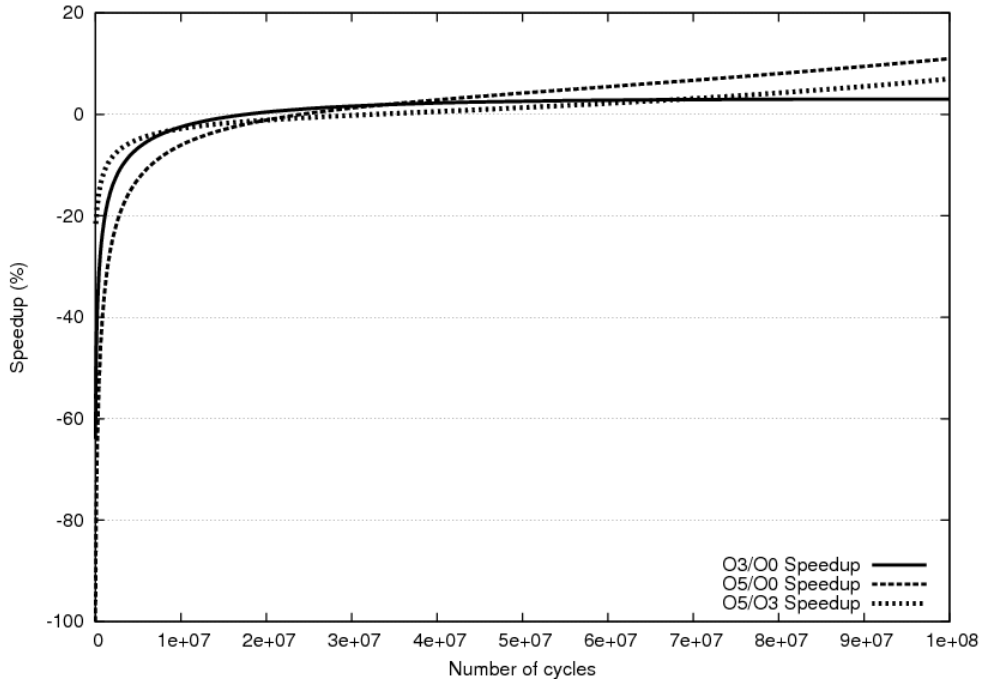

Figure 6.19: `factorial` speedup for `S=79` and `R=4`

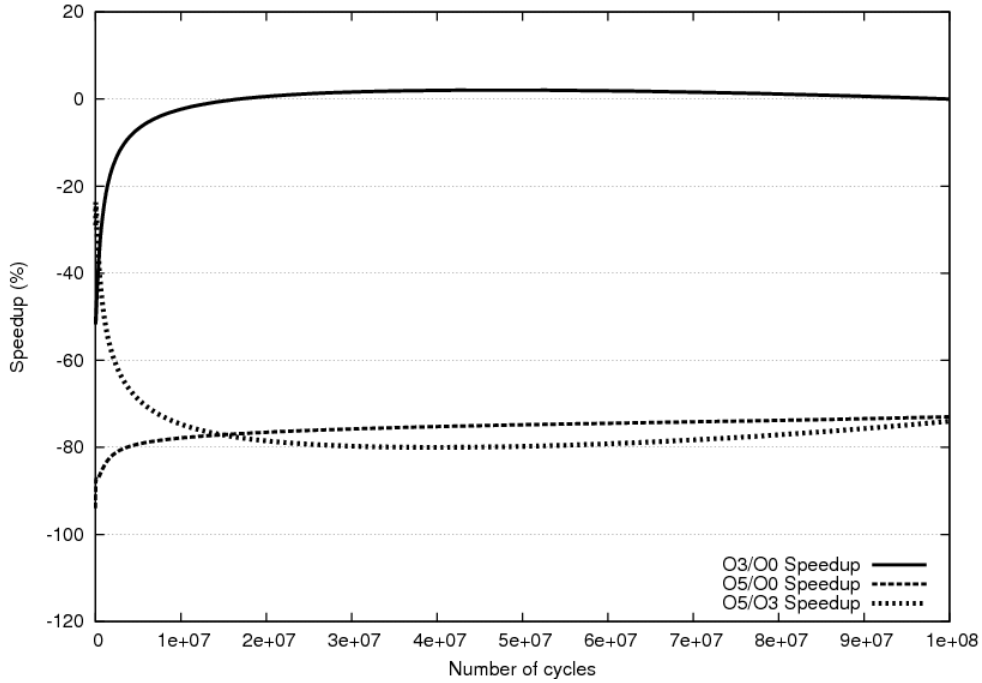

Figure 6.20: `factorial` speedup for `S=79` and `R=5`

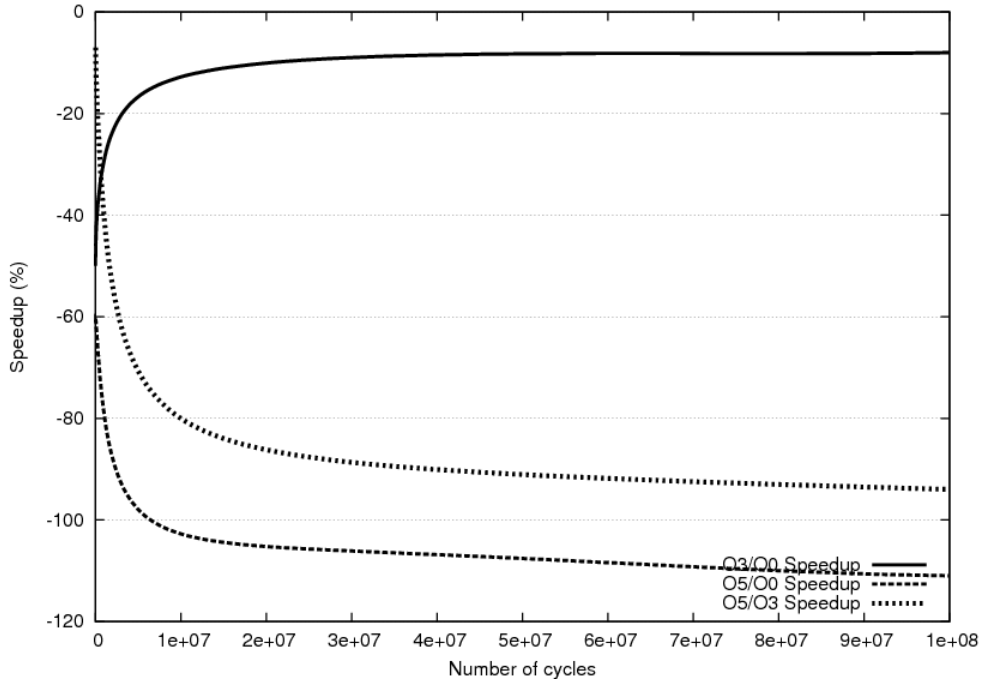

Figure 6.21: `fibonacci` speedup for `S=79` and `R=10`

can be obtained here strongly depends on the algorithm used. In this case the main advantage does not only lay in the fact that can perform a more aggressive optimization of the code, but also that we can call a function with one parameter less because it has been specialized and "translated" in a constant value inside the specialized method.

Consider now the `fibonacci` benchmark. You can see the numeric results in table B and the specialization hints in table A. What you can see here is that all the speed up is negative: this means the it is not worthwhile applying specialization to this program but, on the contrary, if you apply the specialization process you will obtain always bad results.

In figures 6.21, 6.22 and 6.23 you can see these negative speedups. The reason of this bad behaviour of the specializer lays in the fact that the program has been built in order to do not be specializable at all because it forces the specializer to contiguously perform specialization and despecialization of a method.

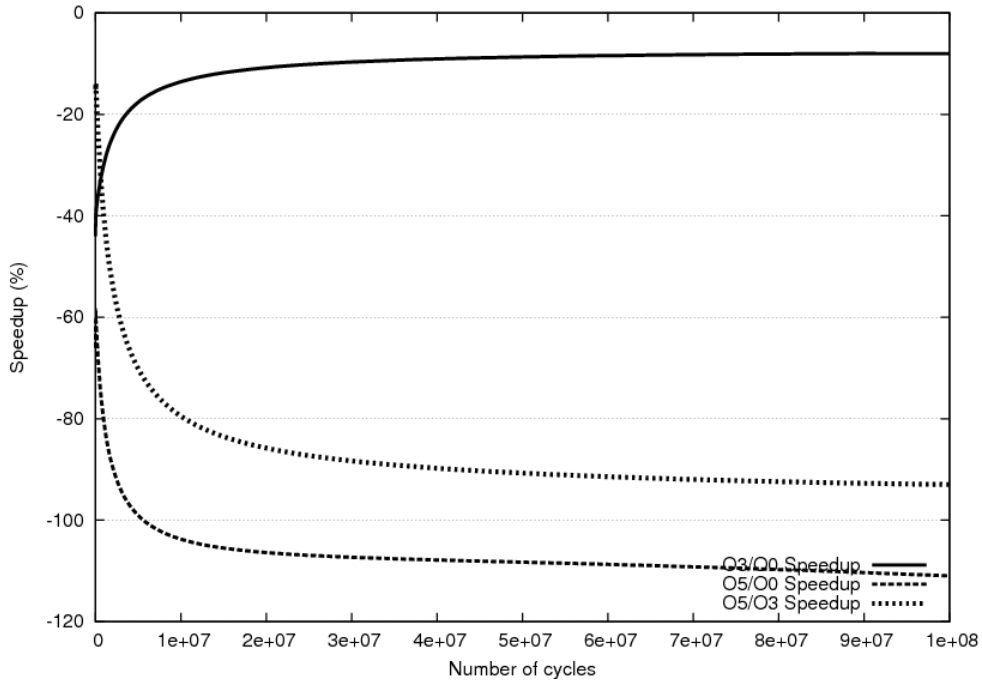

Figure 6.22: `fibonacci` speedup for `S=79` and `R=11`

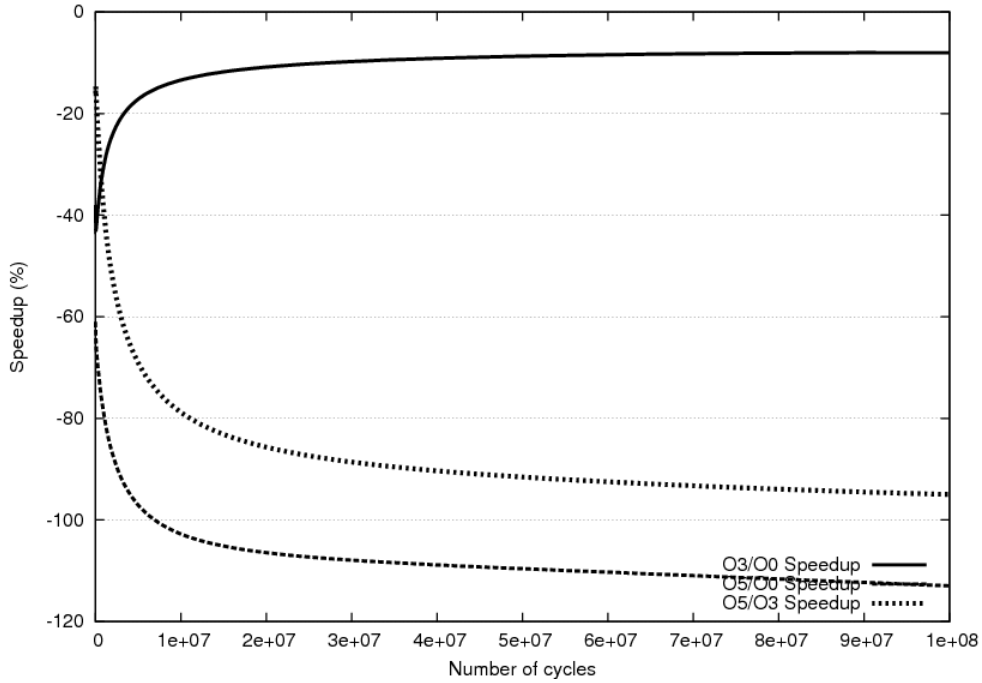

Figure 6.23: `fibonacci` speedup for `S=79` and `R=12`

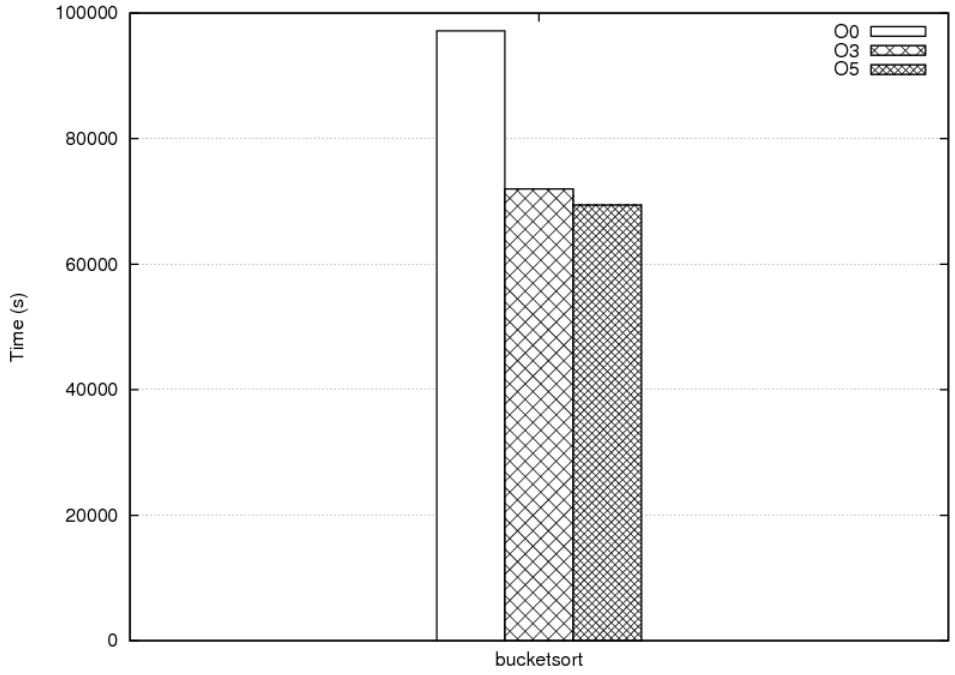

Figure 6.24: Best time performances for `bucketsort`

A speech by itself deserve the `decoder` benchmark. As you can see in the specialization hint table A only two methods could be worthwhile specializing. Anyway, if you look at the numeric result table B you can see that we have negative speed up for the three speed-ups described above. This demonstrate how strongly connected is the specialization process to the optimization process: if the optimization process does not generate optimized code for the program neither will do the specializer.

6.3.1 Best execution results

As final note on the result presentation a graph that shows the best obtained result for each benchmark will be presented in figures 6.24 and 6.25. Anyway, for the `decoder` and the `fibonacci` benchmarks that does not generate any suitable result the worst cases has been plotted.

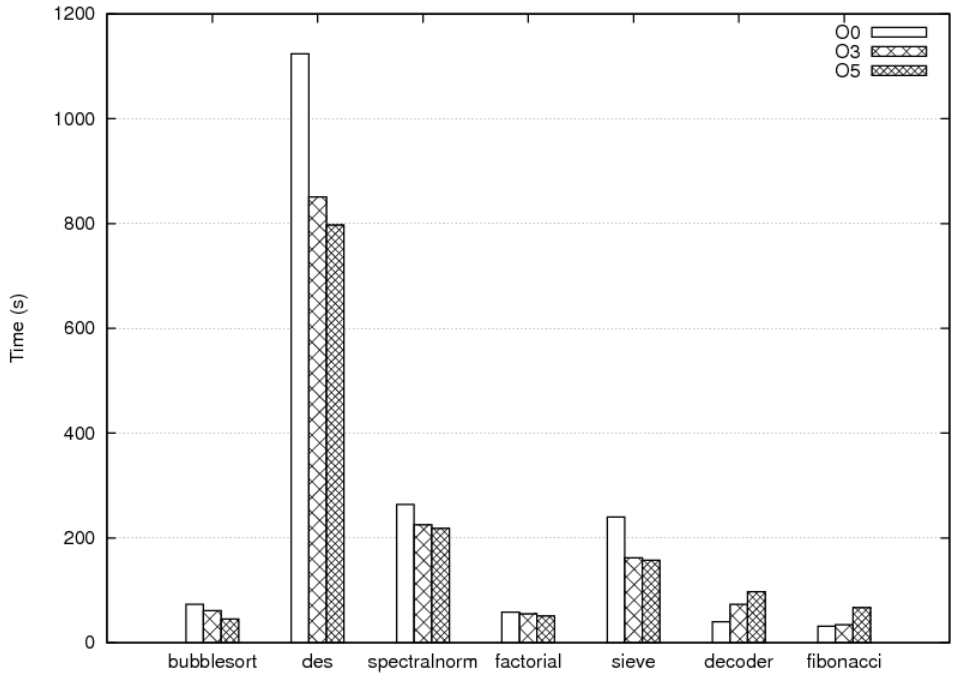

Figure 6.25: Best time performances for `bubblesort`, `des`, `spectralnorm`, `factorial`, `sieve`, `decoder` and `fibonacci`

Chapter 7

Conclusions and future work

The work described in this dissertation aims at opening the way for the exploration of the partial evaluation optimization at run-time inside a just-in-time distributed compiler col the Common Intermediate Len gauge defined by the ECMA-355 specification.

The main contribution lays in the implementation and the experimental analysis of such a technique using the three-way profiling. Other authors had already proposed the use of profiling in conjunction with the specialization process [4, 6, 44, 5], but where they use just a single possible heavy profiler, in this thesis four different profiler has been actually used: the zero profiler, the first profiler, the second profiler and the undispatching profiler.

Moreover, in the most of the cited and already developed and studied specialization the profiling phase is executed off-line in order to avoid the profiling cost from the specialization cost. On the contrary in this implementation the profiling has been performed on-line, i.e. at runtime. The possible cost of specialization has been partially avoided with the modularization of the profiling infrastructure and trying to execute the profiling only when needed.

Another contribution lays in the fact that not many studies has been made on method despecialization. This process can be quite complicated because it is not simple find when it is worthwhile keep specializing or when it is not and the specializer must the remove the specialized instruction injected inside the code.

Here a brief analysis on this topic has been made even if it could be improved in future works.

Through this optimization technique a speedup ranging between 2% and 38% has been possible. These results are comparable with the one defined in [3, 4]. Anyway more improvements should be made possible through the use of more aggressive code optimization and the use of a static compiler that automatically generates the required specialization hints. The use of such a compiler should improve further the specialization process since it could perform more aggressive analysis on the code that the *full specialization mode* described in section 6.1 can not obviously perform.

These encouraging results lead to consider the use of further improvement at the specialization process. This could be made possible by the introduction of the specialization window described in section 4.5.3. Other improvements would possibly introduce a more fine grained selection of which call site are worthwhile specializing: it could be useful not only select which method specialization but also it would be interesting detect which call sites (possibly not all) are useful specializing and which are not.

Appendix A

Experimental Results Details: Expression Hints

In this appendix the hint expressions, generated by the operating mode described in section 6.1, will be presented. Sometimes deafult values will be required. These values are the ones defined inside the C code[1]. A particular notation will be used in order to refer to the profilers and the dispatchers:

Notation	Means
0P	Zero Profiler
1P	First Profiler
2P	Second Profiler
TD	Type Dispatcher
D	Dispatcher

Table A.1: `bucketsort` hints: Parameters and Expressions

Method Signature	Parameters		Expressions
System.Void bucketsort(System.Int32[], System.Int32, System.Int32[])	0P Static Th.	DEFAULT	`1 LENGTH AND 3 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	3	
	2P Memory	96	
	2P Static Th.	0.20	
	D Remove Th.	10	

[1]See file `libiljitirprofiler/src/irspecializer.h`

Table A.2: `bubblesort` hints: Parameters and Expressions

Method Signature	Parameters		Expressions
System.Void bubblesort(System.Int32[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	3	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	

Table A.3: `des` hints: Parameters and Expressions

Method Signature	Parameters		Expressions
System.UInt64 PackKey8(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	
System.UInt64 PackKey16(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	
System.UInt64 PackKey24(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	
System.UInt64 PackKey32(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	
System.UInt64 PackKey40(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	

Continued on next page

Table A.3 *continued* – `des` hints: Parameters and Expressions

Method Signature	Parameters		Expressions
System.UInt64 PackKey48(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	
System.UInt64 PackKey64(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	
System.UInt64 PackKey72(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	
System.UInt64 PackKey(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	

Table A.4: `decoder` hints: Parameters and Expressions

Method Signature	Parameters		Expressions
System.UInt32 showBits(System.UInt32, stream *)	0P Static Th.	DEFAULT	`1 STATIC`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	32	
	2P Static Th.	0.3	
	D Remove Th.	10	
System.UInt32 flushBits(System.UInt32, stream *)	0P Static Th.	DEFAULT	`1 STATIC`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	32	
	2P Static Th.	0.3	
	D Remove Th.	10	

Table A.5: `spectralnormal` hints: Parameters and Expressions

Method Signature	Parameters		Expressions
System.Void MultiplyAtv(System.Int32, System.Double[], System.Double[])	0P Static Th.	DEFAULT	`2 LENGTH AND 3 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	`1 STATIC`
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.2	
	D Remove Th.	10	
System.Void MultiplyAv(System.Int32, System.Double[], System.Double[])	0P Static Th.	DEFAULT	`2 LENGTH AND 3 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	`1 STATIC`
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.2	
	D Remove Th.	10	

Table A.6: `sieve` hints: Parameters and Expressions

Method Signature	Parameters		Expressions
System.Int32 sieve(System.Byte[])	0P Static Th.	DEFAULT	`1 LENGTH`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	100	

Table A.7: `factorial` hints: Parameters and Expressions

Method Signature	Parameters		Expressions
System.UInt64 factorial(System.Int32)	0P Static Th.	DEFAULT	`1 STATIC`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	

Table A.8: `fibonacci` hints: Parameters and Expressions

Method Signature	Parameters		Expressions
System.UInt64 fibonacci(System.Int32)	0P Static Th.	DEFAULT	`1 STATIC`
	0P Memory	DEFAULT	
	TD Remove Th.	DEFAULT	
	1P Th	10	
	2P Memory	96	
	2P Static Th.	0.3	
	D Remove Th.	10	

Appendix B

Experimental Results Detail: Numeric Results

Table B.1: `bucketsort` numeric results for $S = 50$

R	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
2	2500	0,268	0,620	0,923	-131,34	-244,4	-48,87
	10000	0,410	0,673	0,998	-64,15	-143,41	-48,29
	62500	1,395	1,523	1,885	-9,18	-35,13	-23,77
	250000	5,287	4,682	5,053	11,44	4,43	-7,92
	562500	11,712	9,893	10,512	15,53	10,24	-6,26
	1000000	21,537	17,479	17,183	18,84	20,22	1,70
	2250000	51,826	37,979	37,353	26,72	27,93	1,65
	4000000	80333	67987	64991	15,37	19,1	4,41

R	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
3	2500	0,285	0,632	0,972	-121,44	-240,57	-53,80
	10000	0,415	0,752	1,095	-81,2	-163,86	-45,61
	62500	1,673	1,793	2,379	-7,17	-42,2	-32,68
	250000	5,614	4,823	5,117	14,09	8,85	-6,10
	562500	11,679	9,963	10,327	14,7	11,58	-3,65
	1000000	21,102	17,648	17,373	16,37	17,67	1,56
	2250000	56,368	41,172	40,011	26,96	29,02	2,82
	4000000	97192	71968	69426	25,95	28,57	3,53

R	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
4	2500	0,239	0,594	0,803	-148,54	-235,98	-35,19
	10000	0,435	0,695	1,301	-59,77	-199,08	-87,19
	62500	1,701	1,715	5,019	-0,82	-195,06	-192,65
	250000	5,984	5,209	16,018	12,95	-167,67	-207,50
	562500	13,301	11,032	32,503	17,06	-144,36	-194,62
	1000000	23,523	19,401	55,822	17,52	-137,31	-187,72
	2250000	56,044	42,872	127,632	23,5	-127,74	-197,71
	4000000	106251	79098	228426	25,56	-114,9	-188,79

Table B.2: `bubblesort` numeric results for $S = 50$

R	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
2	2500	0,240	0,400	0,490	-66,67	-104,17	-22,50
	10000	0,430	0,560	0,660	-30,23	-53,49	-17,86
	62500	1,740	1,850	1,610	-6,32	7,47	12,97
	250000	6,320	5,920	5,070	6,33	19,78	14,36
	562500	14,050	12,921	10,860	8,04	22,71	15,95
	1000000	24,770	22,531	18,959	9,04	23,46	15,85
	2250000	68,880	57,352	45,080	16,74	34,55	21,40
R	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
3	2500	0,260	0,400	0,500	-53,85	-92,31	-25,00
	10000	0,430	0,560	0,660	-30,23	-53,49	-17,86
	62500	1,750	1,730	1,640	1,14	6,29	5,20
	250000	6,490	6,010	5,210	7,4	19,72	13,31
	562500	14,370	13,180	11,400	8,28	20,67	13,50
	1000000	25,300	23,041	19,601	8,93	22,53	14,93
	2250000	73,125	61,315	45,509	16,15	37,77	25,78
R	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
4	2500	0,260	0,420	0,580	-61,54	-123,08	-38,10
	10000	0,540	0,650	0,940	-20,37	-74,07	-44,62
	62500	2,390	2,300	3,290	3,77	-37,66	-43,04
	250000	9,150	8,110	10,480	11,37	-14,53	-29,22
	562500	20,161	17,961	22,150	10,91	-9,87	-23,33
	1000000	35,299	31,550	38,160	10,62	-8,11	-20,95
	2250000	79,180	73,092	82,982	7,69	-4,8	-13,53

Table B.3: `des` numeric results

E	I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
100	100	1,270	1,760	2,990	-38,58	-135,43	-69,89
250	250	6,940	6,520	8,020	6,05	-15,56	-23,01
500	500	27,071	23,350	23,210	13,75	14,26	0,60
750	750	59,831	54,136	52,541	9,52	12,18	2,95
1000	1000	114,478	92,240	91,276	19,43	20,27	1,04
1500	1500	265,581	213,579	204,120	19,58	23,14	4,43
2000	2000	463,130	368,451	349,584	20,44	24,52	5,12
3000	3000	1124,242	851,520	797,776	24,26	29,04	6,31

Table B.4: `spectralnormal` numeric results

N	C	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
300	100	2,640	2,480	2,610	6,06	1,14	-5,24
	250	6,823	5,950	5,910	12,79	13,38	0,67
	500	12,797	11,110	10,943	13,19	14,49	1,50
	750	20,428	17,800	17,350	12,86	15,07	2,53
	1000	44,943	39,199	38,206	12,78	14,99	2,53
	1500	93,227	81,435	79,258	12,65	14,98	2,67

N	C	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
400	100	4,647	4,190	4,207	9,83	9,47	-0,41
	250	11,380	9,847	9,803	13,47	13,86	0,45
	500	22,567	19,538	18,998	13,42	15,82	2,77
	750	36,091	31,203	30,342	13,54	15,93	2,76
	1000	80,397	68,777	67,161	14,45	16,46	2,35
	1500	166,899	141,964	138,090	14,94	17,26	2,73

N	C	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
500	100	7,240	6,387	6,390	11,78	11,74	-0,05
	250	17,730	15,280	15,047	13,82	15,13	1,52
	500	35,203	30,083	29,470	14,54	16,29	2,04
	750	57,433	48,995	47,585	14,69	17,15	2,88
	1000	128,624	109,771	106,744	14,66	17,01	2,76
	1500	264,189	225,541	218,932	14,63	17,13	2,93

N	C	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
600	100	10,350	9,000	8,890	13,04	14,11	1,22
	250	25,441	21,791	21,400	14,35	15,88	1,79
	500	48,991	41,050	40,109	16,21	18,13	2,29
	750	82,595	68,570	66,761	16,98	19,17	2,64
	1000	174,720	145,565	141,337	16,69	19,11	2,91

N	C	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
700	100	14,069	11,970	11,890	14,92	15,49	0,67
	250	34,710	29,441	28,820	15,18	16,97	2,11
	500	69,653	57,998	56,707	16,73	18,59	2,23
	750	112,122	92,842	90,788	17,19	19,03	2,21
	1000	239,924	198,364	193,630	17,32	19,30	2,39

N	C	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
800	100	18,089	15,520	15,360	14,20	15,09	1,03
	250	44,979	38,149	37,681	15,18	16,23	1,23
	500	90,672	75,193	73,717	17,07	18,70	1,96
	750	139,270	114,885	112,594	17,51	19,15	1,99
	1000	310,804	255,826	249,730	17,69	19,65	2,38

Table B.5: `factorial` numeric results for $S = 79$

S	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
2	2500	0,19	0,29	0,35	-52,63	-84,21	-20,69
	10000	0,18	0,28	0,35	-55,56	-94,44	-25,00
	62500	0,23	0,32	0,39	-39,13	-69,57	-21,87
	250000	0,38	0,47	0,54	-23,68	-42,11	-14,89
	562500	0,61	0,7	0,77	-14,75	-26,23	-10,00
	1000000	0,95	1,03	1,1	-8,42	-15,79	-6,80
	2250000	1,92	1,97	2,04	-2,60	-6,25	-3,55
	4000000	3,24	3,28	3,42	-1,23	-5,56	-4,27
	9000000	7,08	7,04	7,13	0,56	-0,71	-1,28
	25000000	19,37	19,05	19,36	1,65	0,05	-1,63
	56250000	43,42	42,77	42,79	1,49	1,45	-0,04
	100000000	79,13	75,97	73,54	4,00	7,07	3,19
S	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
3	2500	0,17	0,29	0,35	-70,59	-105,88	-20,69
	10000	0,18	0,3	0,36	-66,67	-100,00	-20,00
	62500	0,22	0,32	0,38	-45,45	-72,73	-18,75
	250000	0,34	0,44	0,5	-29,41	-47,06	-13,64
	562500	0,54	0,63	0,68	-16,67	-25,93	-7,94
	1000000	0,81	0,91	0,94	-12,35	-16,05	-3,30
	2250000	1,62	1,67	1,7	-3,09	-4,94	-1,80
	4000000	2,76	2,77	2,74	-0,36	0,72	1,08
	9000000	5,97	5,88	5,71	1,51	4,36	2,89
	25000000	16,35	15,8	15,22	3,36	6,91	3,67
	56250000	36,43	35,24	33,86	3,26	7,06	3,92
	100000000	65,5	61,57	57,83	5,99	11,71	6,08
S	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
4	2500	0,17	0,28	0,34	-64,71	-100,00	-21,43
	10000	0,19	0,29	0,36	-52,63	-89,47	-24,14
	62500	0,22	0,31	0,38	-40,91	-72,73	-22,58
	250000	0,33	0,43	0,48	-30,30	-45,45	-11,63
	562500	0,5	0,58	0,65	-16,00	-30,00	-12,07
	1000000	0,75	0,84	0,91	-12,00	-21,33	-8,33
	2250000	1,47	1,53	1,59	-4,08	-8,16	-3,92
	4000000	2,48	2,49	2,56	-0,40	-3,23	-2,81
	9000000	5,34	5,25	5,31	1,69	0,56	-1,14
	25000000	14,55	14,09	14,13	3,17	2,89	-0,29
	56250000	32,74	31,46	31,41	3,91	4,06	0,16
	100000000	58,07	55,94	51,64	3,67	11,07	7,68

Continued on the next page

Table B.5 *continued* – `factorial` numeric results for $S = 79$

S	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
5	2500	0,18	0,27	0,35	-50,00	-94,44	-29,63
	10000	0,19	0,3	0,35	-57,89	-84,21	-16,67
	62500	0,24	0,35	0,43	-45,83	-79,17	-22,86
	250000	0,34	0,49	0,71	-44,12	-108,82	-44,90
	562500	0,49	0,57	0,85	-16,33	-73,47	-49,12
	1000000	0,72	0,79	1,29	-9,72	-79,17	-63,29
	2250000	1,38	1,45	2,51	-5,07	-81,88	-73,10
	4000000	2,33	2,34	4,17	-0,43	-78,97	-78,21
	9000000	5,03	4,92	8,86	2,19	-76,14	-80,08
	25000000	13,68	13,18	23,99	3,65	-75,37	-82,02
	56250000	30,57	29,35	53,65	3,99	-75,49	-82,78
	100000000	53,67	53,44	93,08	0,43	-73,43	-74,18

Table B.6: `sieve` numeric results for $S = 1000$

R	C	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
2	2500	0,32	0,41	0,54	-28,12	-68,75	-31,71
	10000	0,7	0,72	0,82	-2,86	-17,14	-13,89
	62500	3,42	2,82	2,83	17,54	17,25	-0,35
	250000	13,22	10,35	9,97	21,71	24,58	3,67
	562500	29,35	23,1	21,83	21,29	25,62	5,50
	1000000	51,73	40,5	38,48	21,71	25,62	4,99
	2250000	127,93	94,06	86,85	26,48	32,11	7,67
	4000000	240,36	162,69	157,37	32,32	34,53	3,27

R	C	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
3	2500	0,32	0,41	0,54	-28,13	-68,75	-31,71
	10000	0,71	0,71	0,83	0,00	-16,90	-16,90
	62500	3,42	2,81	2,82	17,84	17,54	-0,36
	250000	13,14	10,36	9,98	21,16	24,05	3,67
	562500	29,29	22,93	21,84	21,72	25,43	4,75
	1000000	52,69	40,53	38,5	23,08	26,93	5,01
	2250000	136,17	91,95	87	32,47	36,11	5,38
	4000000	253,76	164,72	154,9	35,09	38,96	5,96

R	C	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
4	2500	0,32	0,42	0,56	-31,25	-75,00	-33,33
	10000	0,7	0,71	1,28	-1,43	-82,86	-80,28
	62500	3,43	2,83	5,6	17,49	-63,27	-97,88
	250000	13,13	10,36	19,51	21,10	-48,59	-88,32
	562500	29,37	22,94	40,47	21,89	-37,80	-76,42
	1000000	52,21	40,74	69,19	21,96	-32,53	-69,84
	2250000	133,55	89,77	144,81	32,78	-8,43	-61,31
	4000000	250,18	170,5	270,28	31,85	-8,03	-58,53

Table B.7: `fibonacci` numeric results for $S = 79$

S	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
10	2500	0,18	0,27	0,29	-50,00	-61,11	-7,41
	10000	0,18	0,26	0,29	-44,44	-61,11	-11,54
	62500	0,2	0,28	0,31	-40,00	-55,00	-10,71
	250000	0,26	0,35	0,45	-34,62	-73,08	-28,57
	1000000	0,5	0,6	0,98	-20,00	-96,00	-63,33
	2250000	0,9	1,02	1,83	-13,33	-103,33	-79,41
	4000000	1,45	1,63	3,01	-12,41	-107,59	-84,66
	9000000	3,05	3,36	6,33	-10,16	-107,54	-88,39
	25000000	8,32	8,88	16,88	-6,73	-102,88	-90,09
	56250000	18,08	19,74	38,09	-9,18	-110,66	-92,95
	100000000	32,01	34,73	67,6	-8,50	-111,18	-94,63
S	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
11	2500	0,18	0,26	0,3	-44,44	-66,67	-15,38
	10000	0,19	0,26	0,29	-36,84	-52,63	-11,54
	62500	0,19	0,27	0,31	-42,11	-63,16	-14,81
	250000	0,26	0,35	0,45	-34,62	-73,08	-28,57
	1000000	0,49	0,61	0,98	-24,49	-100,00	-60,66
	2250000	0,9	1,02	1,83	-13,33	-103,33	-79,41
	4000000	1,45	1,63	3	-12,41	-106,90	-84,05
	9000000	3,03	3,36	6,31	-10,89	-108,25	-87,80
	25000000	8,11	8,86	16,85	-9,25	-107,76	-90,17
	56250000	18,04	19,6	37,66	-8,66	-108,77	-92,14
	100000000	31,96	34,83	67,55	-8,98	-111,37	-93,95
S	C=E×I	-O0 (s)	-O3 (s)	-O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
12	2500	0,18	0,25	0,29	-38,89	-61,11	-16,00
	10000	0,18	0,27	0,3	-50,00	-66,67	-11,11
	62500	0,19	0,27	0,32	-42,11	-68,42	-18,52
	250000	0,26	0,35	0,43	-34,62	-65,38	-22,86
	1000000	0,5	0,6	0,97	-20,00	-94,00	-61,67
	2250000	0,9	1,02	1,81	-13,33	-101,11	-77,45
	4000000	1,44	1,64	2,99	-13,89	-107,64	-82,32
	9000000	3,04	3,35	6,3	-10,20	-107,24	-88,06
	25000000	8,09	8,84	16,84	-9,27	-108,15	-90,49
	56250000	17,98	19,56	37,92	-8,78	-110,89	-93,86
	100000000	31,83	34,63	67,81	-8,80	-113,04	-95,82

Table B.8: `decoder` numeric results

O0 (s)	O3 (s)	O5 (s)	O3/O0 (%)	O5/O0 (%)	O5/O3 (%)
41,75	74,86	93,93	-78,43	-138,31	-33,56

List of Figures

List of Tables

Bibliography

[1] Charles Consel and François Noël. A general approach for run-time specialization and its application to c. In POPL, pages 145–156, 1996.

[2] Karina Olmos and Eelco Visser. Turning dynamic typing into static typing by program specialization in a compiler front-end for octave. In SCAM, pages 141–150, 2003.

[3] Ulrik Schultz and Charles Consel. Automatic program specialization for java. ACM Transactions on Programming Languages and Systems, 25:2003, 2003.

[4] Ajeet Shankar, S. Subramanya Sastry, Rastislav Bodík, and James E. Smith. Runtime specialization with optimistic heap analysis. In OOPSLA, pages 327–343, 2005.

[5] Eui-Young Chung, Luca Benini, Giovanni De Micheli, Gabriele Luculli, and Marco Carilli. Value-sensitive automatic code specialization for embedded software. IEEE Trans. on CAD of Integrated Circuits and Systems, 21(9):1051–1067, 2002.

[6] Robert Muth, Scott A. Watterson, and Saumya K. Debray. Code specialization based on value profiles. In SAS, pages 340–359, 2000.

[7] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In CGO, pages 198–208, 2007.

[8] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: a tool for automating selective dynamic compilation. In MICRO, pages 291–302, 2000.

[9] Hidehiko Masuhara and Akinori Yonezawa. A portable approach to dynamic optimization in run-time specialization. New Generation Comput., 20(1):101–124, 2001.

[10] Charles Consel, Luke Hornof, François Noël, Jacques Noyé, and Nicolae Volansche. A uniform approach for compile-time and run-time specialization. In Dagstuhl Seminar on Partial Evaluation, pages 54–72, 1996.

[11] Gustavo J. Bobeff and Jacques Noyé. Component specialization. In PEPM, pages 39–50, 2004.

[12] Gustavo Bobeff and Jacques Noyè. Molding components using program specialization techniques. In In WCOP 2003, Eighth International Workshop on ComponentOriented Programming, 2003.

[13] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In PEPM, pages 15–26, 2004.

[14] Andrew W. Appel. Modern Compiler Implementation in Java, 2nd edition. Cambridge University Press, 2002.

[15] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers, Principles, Techniques, and Tools. Addison-Wesley, 1986.

[16] Y. N. Srikant and Priti Shankar, editors. The Compiler Design Handbook: Optimizations and Machine Code Generation. CRC Press, 2002.

[17] Stefano Crespi Reghizzi. Linguaggi Formali e Compilazione. Pitagora Editrice Bologna, 2006.

[18] Psyco sourceforge page. http://psyco.sourceforge.net/.

[19] Phoenix Tempo Home Page. http://phoenix.labri.fr/software/tempo/.

[20] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. The benefits and costs of dyc's run-time optimizations. ACM Trans. Program. Lang. Syst., 22(5):932–972, 2000.

[21] Evelyn Duesterwald. Dynamic compilation. In The Compiler Design Handbook, pages 739–762. 2002.

[22] Giovanni Agosta and Stefano Crespi Reghizzi. Dynamic look ahead compilation. 2008.

[23] Microsoft .NET web site. http://msdn.microsoft.com/netframework/default.aspx.

[24] Barry Cornelius. Comparing .net with java. Technical report, University of Oxford, 2002.

[25] Scala language web site. http://www.scala-lang.org/.

[26] Groovy web site. http://groovy.codehaus.org/.

[27] Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. A parallel dynamic compiler for cil bytecode. SIGPLAN Not., 43(4):11–20, 2008.

[28] ILDJIT sourceforge page. http://ildjit.sourceforge.net/.

[29] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. Deli: a new run-time control point. In MICRO, pages 257–268, 2002.

[30] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In PLDI, pages 1–12, 2000.

[31] Mono web page. http://www.mono-project.com.

[32] Portable.NET web site. http://www.dotgnu.org/.

[33] Reynald Affeldt, Hidehiko Masuhara, Eijiro Sumii, and Akinori Yonezawa. Supporting objects in run-time bytecode specialization. In ASIA-PEPM, pages 50–60, 2002.

[34] Bernd Burgstaller, Bernhard Scholz, and Johann Blieberger. Symbolic analysis of imperative programming languages. In Lightfoot and Szyperski [45], pages 172–194.

[35] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In LCPC, pages 538–562, 1992.

[36] Graeme D. Cunningham. Expression coverability analysis: Improving code coverage with model checking.

[37] Hiralal Agrawal. Dominators, super blocks, and program coverage. In POPL, pages 25–34, 1994.

[38] BWM garbage collector web site. http://www.hpl.hp.com/personal/hans_boehm/gc.

[39] Libjit documentation. http://www.gnu.org/software/dotgnu/libjit-doc/libjit.html.

[40] Thomas W. Reps and Todd Turnidge. Program specialization via program slicing. In Dagstuhl Seminar on Partial Evaluation, pages 409–429, 1996.

[41] Shimarks 2.0 benchmarks. http://math.nist.gov/scimark2/.

[42] Bruno Richard Preiss. Data Structures and Algorithms with Object-Oriented Design Patterns in C#. 2001.

[43] JavaGrande benchmarks. http://www.javagrande.org/.

[44] Brad Calder, Peter Feller, and Alan Eustace. Value profiling and optimization. J. Instruction-Level Parallelism, 1, 1999.

[45] David E. Lightfoot and Clemens A. Szyperski, editors. Modular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006, Oxford, UK, September 13-15, 2006, Proceedings, volume 4228 of Lecture Notes in Computer Science. Springer, 2006.

[46] Scott Thibault, Charles Consel, Julia L. Lawall, Renaud Marlet, and Gilles Muller. Static and dynamic program compilation by interpreter specialization. Higher-Order and Symbolic Computation, 13(3):161–178, 2000.

[47] Hidehiko Masuhara and Akinori Yonezawa. Generating optimized residual code in run-time specialization. In In International Colloquium on Partial Evaluation and Program Transformation, pages 83–102, 1999.

[48] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In ICCL, pages 132–142, 1998.

[49] Michael D. Smith. Overcoming the challenges to feedback-directed optimization. In Dynamo, pages 1–11, 2000.

[50] Stephen A. Edwards. Using program specialization to speed systemc fixed-point simulation. In PEPM, pages 21–28, 2006.

[51] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. Advanced Linux Programming. New Riders Publishing, 2001.

[52] Romana Baier, Robert Glück, and Robert Zöchling. Partial evaluation of numerical programs in fortran. In PEPM, pages 119–132, 1994.

[53] Lars Ole Andersen. Self-applicable c program specialization. In PEPM, pages 54–61, 1992.

[54] Robert Gluck and Neil D. Jones. Automatic program specialization by partial evaluation: an introduction. In Software Engineering in Scientific Computing, pages 70–77, 1996.

[55] Sandrine Chirokoff, Charles Consel, and Renaud Marlet. Combining program and data specialization. Higher-Order and Symbolic Computation, 12(4):309–335, 1999.

[56] Karoline Malmkjær and Peter Ørbæk. Polyvariant specialisation for higher-order, block-structured languages. In PEPM, pages 66–76, 1995.

[57] Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental run-time specialization for free. In PLDI, pages 281–292, 1999.

[58] Ulrik Pagh Schultz. Black-box program specialization. In ECOOP Workshops, page 187, 1999.