ANNÉE 2018

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

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale MathSTIC

présentée par

Marcelino RODRIGUEZ CANCIO

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Aléatoires ISTIC –
UFR en Informatique et Électronique

Contributions on Approximate Computing Techniques and How to Measure Them

Thèse soutenue à Rennes le 19 Décembre 2017

devant le jury composé de :

Olivier SENTIEYS
Professeur à l'Université de Rennes 1 / *Président*

Julia LAWALL
Directrice de Recherche, Inria/LIP6 / *Rapporteuse*

Douglas SCHMITHD
Professeur à l'Université de Vanderbilt / *Rapporteur*

Romain ROUVOY
Professeur à l'Université de Lille / *Rapporteur*

Benoit BAUDRY
Professeur à KTH Royal Institute of Technology / *Directeur de thèse*

Benoit COMBEMALE
Professeur à l'Université de Toulouse / *Co-directeur de thèse*

The difficult is what takes a little time; the impossible is what takes a little longer.

Fridtjof Nansen

Contents

Introduction in Français

La **Computation Aproximée** fournit un cadre de réflexion qui nous permet de raisonner sur l'exactitude en tant que ressource qui peut être exploitée. Les besoins en ressources informatiques sont constants dans tous les domaines de l'informatique. Cependant, les processeurs plus rapides et les dispositifs de stockage plus gros coûtent plus cher, consomment plus d'énergie et dissipent plus de chaleur. C'est pourquoi les idées d'Approximate Computing séduisent les concepteurs d'une large gamme de plates-formes, des systèmes embarqués aux grands superordinateurs.

De nombreux domaines informatiques sont de nature approximative. Le traitement numérique du signal est un domaine où la précision est perdue dès que le signal est numérisé. Dans le Machine Learning, il n' y a généralement pas de bons ou de mauvais résultats, mais plutôt des résultats plus ou moins précis. L'analyse de Big Data utilise des méthodes statistiques qui tolèrent les imprécisions dans des éléments de données individuels. Les applications dans ces domaines contiennent de nombreuses *Zones de Pardon* qui peuvent endurer des imprécisions, offrant de nombreuses opportunités d'exploiter la précision en tant que ressource. Néanmoins, les développeurs s'efforcent généralement de rendre les programmes plus précis que nécessaire, en gaspillant des ressources qui pourraient être allouées efficacement ailleurs.

Les zones de pardon sont l'élément clé de la Computation Aproximée. Ces zones sont des parties d'un système (code, fils, composants de hardware, données d'entrée/sortie, accès mémoire) qui peuvent subir un certain degré d'approximation.

Contributions Cette thèse propose trois contributions dans le domaine de l'Approximate Computing (i) une optimisation approximative du compilateur, (ii) un outil pour mesurer automatiquement la performance des petits segments de code et (iii) un mécanisme de compression de code qui considère les programmes comme des données de perte.

Approximate Loop Unrolling

Approximate Loop Unrolling est une nouvelle optimisation qui utilise les idées de la Computation Aproximée pour réduire les temps d'exécution et la consommation d'énergie des boucles. L'optimisation combine l'analyse statique et les transformations de code afin d'améliorer la performance des boucles en échange de faibles pertes de précision.

Approximate Unrolling repose sur l'observation que les données telles que les séries temporelles, le son, la vidéo et les images sont souvent représentées comme des tableaux où les emplacements contigus contiennent des valeurs similaires. Comme conséquence de cette similitude voisine, les calculs produisant ces données sont généralement des fonctions lisses localement. En d'autres termes, les calculs produisant ou modifiant des valeurs de tableaux proches représentant ces types de données, produisent souvent des résultats similaires.

Notre technique exploite cette observation en cherchant des boucles où les fonctions sont mappées à des places de tableaux contiguës dans chaque itération. Si le calcul de la fonction est coûteux, nous la remplaçons par des interpolations moins coûteuses des valeurs assignées aux valeurs du tableau voisin. En échange de cette perte de précision, nous obtenons une boucle plus rapide et moins consommatrice d'énergie.

Nous modifions le code source du compilateur C2 d'OpenJDK pour inclure Approximate Unrolling. Nous avons réalisé une série d'expériences avec cette implémentation et un ensemble soigneusement sélectionné de bibliothèques Java du monde réel.

Nos résultats montrent qu'Approximate Unrolling est capable de réduire le temps d'exécution et la consommation d'énergie du CPU pour le code généré d'environ 50% à 110% tout en maintenant la Qualité de Service (QoS) à des niveaux acceptables. Comparativement à la Perforation de Boucle (l'état de l'art de la technique de calcul approximatif pour l'optimisation de boucles), Approximate Unrolling a mieux préservé la précision dans 76% des cas et a soulevé moins d'exceptions fatales.

Primer

Les techniques de compression de code ont un effet positif dans des domaines économiques importants tels que les systèmes embarqués et Internet of Things/Wireless Sensor Networks (IoT/WSN). Une taille de code plus petite réduit les besoins en mémoire des systèmes embarqués. Cela se traduit par des économies considérables en termes de consommation d'énergie et de coûts de fabrication. La compression du code permet également de réduire le trafic réseau, ce qui représente une consommation d'énergie énorme pour les périphériques IoT/WSN. Par conséquent, lors de la reprogrammation à distance de dispositifs IoT/WSN, un système efficace de compression de code pour le code est considéré comme bénéfique ou même nécessaire.

Cette thèse présente **Primer**, le premier (à notre connaissance) algorithme de compression lossy pour les instructions de l'assembleur ARM. Primer est basé sur deux observations. La première observation est que si certains bits d'un programme sont supprimés, ils peuvent être déduits en retour, en sachant comment un programme doit se comporter. La deuxième observation est l'existence de zones de pardon dans les programmes. Primer va exploiter les zones de tolérance pour compresser les programmes d'une manière lossy : le programme décompressé peut contenir un certain degré de dégradation sous forme de flips bit sur ses instructions assembleur et terminer son exécution correctement.

Primer fonctionne en enlevant jusqu' à 20% du total des bits du programme. Cette suppression est faite en utilisant une stratégie soigneusement conçue basée sur l'Algorithme d'Emballage Successif (SPA), qui sélectionne les bits à enlever d'une manière qui maximise les chances de récupérer les bits manquants. Résultat de la suppression des bits, il y a maintenant 2^k combinaisons de valeurs possibles. Ensuite, Primer trie toutes les combinaisons de valeurs possibles par leur probabilité d'être la combinaison présente dans le programme original. La position dans cet ordre est utilisée pour encoder chaque combinaison. L'intuition ici est que la combinaison du programme original aura un petit numéro de encodage, nécessitant moins de mémoire pour représenter. Jusqu' à cette phase, Primer est un algorithme sans perte et peut être interrompu ici. Si l'algorithme est autorisé à continuer, il utilise une fonction d'évaluation de la qualité de service (QoS) pour déterminer quelles encodages plus petites (codage de programmes similaires) peuvent être utilisées sans affecter la QoS. Notez que les encodages plus petites exigent moins de mémoire, donc l'utilisation d'un programme similaire au lieu de l'original peut entraîner une compression supplémentaire.

Nous avons évalué Primer en utilisant `GZip` comme base de référence. Nos expériences ont consisté à comprimer un ensemble de programmes 32 bits ARMv5 en utilisant `GZip` comme base de référence et à comparer les taux de compression avec ceux obtenus en utilisant Primer. Nos résultats montrent que Premier a amélioré jusqu' à 10

AutoJMH

La plupart des auteurs du domaine de l'Approximate Computing utilisent des benchmarks tels que SPEC, SciMark et PARSEC pour évaluer les gains d'accélération obtenus avec leurs techniques. Les benchmarks sont la façon la plus simple de tester les performances. Elles sont déjà construites ; elles sont entretenues et révisées par des spécialistes de la performance, ce qui facilite la répétition des expériences. C'est pourquoi les tests de référence sont largement acceptés par la communauté scientifique. Cependant, les benchmarks sont de grands programmes avec des milliers ou des millions de lignes de code. Par conséquent, lorsque des optimisations sont appliquées à certaines parties de ces programmes, l'accélération perçue des performances dépend fortement de l'impact de la zone optimisée sur le système, plutôt que de l'effet de la technique elle-même. Pour étudier l'impact d'une technique indépendamment d'un système, les ingénieurs en performance de l'industrie ont proposé une technique différente (et complémentaire) appelée Microbenchmarks. Ce type d'essais de performance permet d'effectuer les meilleurs tests de performance du grain (p. ex., essai de la performance d'une boucle unique ou d'une assignation variable).

Cependant, le développement des microbenchmarks reste encore un métier que seuls quelques experts maîtrisent. En particulier, le manque de soutien des outils empêche une adoption plus large du microbenchmarking. Les ingénieurs qui conçoivent des microbenchmark font très souvent deux erreurs : ils oublient de concevoir les essais d'une manière qui empêche le JIT d'effectuer l'élimination des codes morts et les plis/propagations constantes (CF/CP). Par conséquent, le microbenchmark est soumis à des optimisations différentes de celles du segment d'origine analysé et le temps mesuré ne reflète pas la prise en compte de ce segment dans le contexte de l'application plus large.

Dans cette thèse, nous proposons de générer automatiquement des microbenchmarks Java à partir d'un segment spécifique d'une application Java. Les tests générés sont garantis exempts de code mort et empêchent les CF/CP.

Nous évaluons la qualité des microbenchmarks générés automatiquement en recréant des microbenchmarks manuscrits par des experts en performance. Les microbenchmarks générés automatiquement mesurent les mêmes temps que les microbenchmarks écrits par les experts de l'HJM. De plus, nous demandons aux ingénieurs Java professionnels sans expérience en évaluation de performance de construire des microbenchmarks. Tous ces tests conduisent à des mesures faussées dues à des décisions naïves lors de la conception des benchmarks, alors que celles générées par notre outil ont permis d'éviter toutes ces erreurs.

Conclusions et Travaux Futurs

La Computation Aproximée est un vaste champ d'études qui propose un nouveau point de vue sur l'exactitude. Cette thèse proposait trois contributions au domaine (i) Approximate Unrolling : une optimisation de compilateur indépendant de la machine, (ii) AutoJMH : un outil pour mesurer l'accélération des segments de code faisant l'objet d'une approximation et (iii) Primer : le premier algorithme de perte pour les instructions d'assembleur.

Notre travail futur consistera à ajouter à l'implémentation d'Approximate Unroroll un moyen de prédire les pertes de précision. L'implémentation est actuellement capable de prédire seulement quelles boucles obtiendront des gains d'accélération grâce aux transformations. Une autre direction est AutoJHM. Bien qu'AutoJMH est capable d'éviter plusieurs erreurs courantes commises par des ingénieurs sans expérience en microbenchmarking, il est encore loin d'être capable d'automatiser complètement toutes les situations potentielles qui peuvent survenir lors de la conception d'un test. Par conséquent, nous envisageons d'ajouter la prise en charge d'une gamme plus large de pièges de microbenchmarking, ce qui rend l'outil plus intelligent. Enfin, bien que Primer est capable de fournir des taux de compression prometteuses, nous croyons qu'il reste encore beaucoup à explorer dans cette nouvelle direction. En outre, des questions telles que la vitesse de compression et la consommation d'énergie de l'algorithme sont restées inexplorées.

Chapter 1

Introduction

1.1 Context

Approximate Computing provides a thought framework that allow us to reason about accuracy as a resource. This clever view produces yet another insight: just like any asset, accuracy can be traded in exchange for other resources. Indeed, previous works in the area show promising results, as they obtain considerable improvements in execution times [SDMHR11], memory usage [FRPVTCS13] and energy consumption [SDH$^+$14] in exchange for small imprecisions in programs. These works have discovered the existence of Forgiving Zones where accuracy can be tunned to make a program faster or more energy-efficient.

Many computational domains are approximate in nature. Digital Signal Processing is a field where precision is lost as soon as the signal is digitalized. In Machine Learning there are usually no right or wrong outputs, but rather more or less accurate ones. Big Data analysis uses statistical methods that tolerate imprecisions in single elements of data. Applications in these domains contain numerous Forgiving Zones that can endure inaccuracy, providing numerous opportunities to exploit accuracy as a resource. Still, developers usually strive to make programs more accurate than required, wasting resources that could be efficiently allocated elsewhere.

On the other hand, a proper analysis on accuracy requirements might lead to better application capabilities. An example of this are sensors networks that only report data diverging from predicted models, extrapolating the rest [ACDFP09]. Giving up some accuracy, the sensor network extends considerably the life of its power source.

The need for computational resources is constant in all fields of computer science. Yet, faster CPUs and larger storage devices cost more money, consume more energy and dissipate more heat. Hence, the ideas of Approximate Computing are appealing to designers of a wide range of platforms, from embedded systems to large super-computers. Embedded designers are constantly pushing towards better battery life (i.e. energy consumption), lower device cost (which increases profit) and lower operational temperatures (which limits device's location). In the field, these factors are frequently of greater concern than accuracy. On the side of the computational spectrum, the explosion in information generated by the industry has forced mayor companies to constantly invest in more data centers, making the energy bill a cost to consider. The storage requirements [GR11] and energy consumption [Del15] forecast by 2020 has led authors [Mit16, SJLM14] to consider approximation as an alternative, questioning whether buying more data centers alone will cope with the projected space and processing requirements.

Forgiving Zones Forgiving Zones are the key enabler in Approximate Computing. Such zones are parts within a system (code, wires, hardware components, input/output data, memory accesses) that can endure some degree of approximation. During the rest of this report we will use the concept repeatedly and referring to a broad set of elements, such as loops [SDMHR11, BC10], circuit components [RRV+14], input parameters [HSC+11], variables types [BM99, BMM14] and memory access [TZW+15], among others.

1.2 Contributions

This thesis proposes three contributions to the field of Approximate Computing (i) a approximate compiler optimization, (ii) a tool to automatically measure the performance of small code segments and (iii) a code compression mechanism that sees programs as lossy data. The two first contributions (Approximate Unrolling and AutoJMH) are related to programming with approximation, providing a way to approximate and a way to measure the performance impact of approximation, respectively. The third contribution is a compression algorithm that goes beyond the traditional believe that programs must be compressed in a lossless way, showing that lossy techniques can be used to compress program instructions as well.

1.2.1 Approximate Unrolling

To effectively program with approximation, developers must be given language, runtime and compiler support. Ideally, an approximating compiler would be equipped with a large toolbox of machine-independent and machine-dependent approximate optimizations that exploit accuracy-as-a-resource in the widest possible range of situations. This is justified as approximation techniques address only specific forgiving zone types. Unfortunately, the optimization available today to approximate compilers are limited to a tiny set of situations [SBR+15] when compared with traditional optimizing compilers.

This thesis increases the amount of forgiving zones approximating compilers can handle by presenting Approximate Unrolling, a scalar loop optimization that reduces execution time and energy consumption of counted loops mapping a function over the elements of an array. Approximate Unrolling transforms loops similarly to Loop Unrolling. However, unlike its exact counterpart, our optimization does not unroll loops by adding exact copies of the loop's body. Instead, it adds code that interpolates the results of previous iterations.

Approximate Unrolling was implemented in the OpenJDK Hotspot C2 Compiler. Using this modified version of the industry standard compiler, a series of experiments in four real-world, popular Java libraries from various domains were performed. The results show that Approximate Unrolling increased performance and energy savings (110% to 200%) while keeping results within acceptable accuracy bounds. Approximate Unrolling is also compared to Loop Perforation, the state of the art approximate loop approximation. The findings are (i) both optimizations are best used in different scenarios (ii) when both optimizations could be applied, Approximate Unrolling preserved accuracy better in 76% of the cases (iii) Approximate Unrolling is safer, as it caused the programs in our dataset to crash only twice, vs. seven crashes caused by Loop Perforation.

1.2.2 Automatic generation of Java Microbenchmarks

Approximate Computing has two opposing goals: (i) to improve on resource consumption at the expenses of accuracy and (ii) do so while maintaining programs' Quality of Service (QoS). Several works address the problem of measuring QoS [RSA+15, MSHR10, SPM+14]. However, to the best of our knowledge, there is no tool that can automate large scale suites of performance benchmarks.

The thesis contributes AutoJMH, a tool able to create such suites, by generating automatically a suite of performance tests know as Microbenchmarks. Such tests evaluate, in isolation, the execution time of small code segments that play a critical role in large applications. The accuracy of a microbenchmark depends on two critical tasks: to wrap the code segment into a payload that faithfully recreates the execution conditions of the large application; and to build a scaffold that runs the payload a large number of times to get a statistical estimate of the execution time. While recent frameworks such as the Java Microbenchmark Harness (JMH) address the scaffold challenge, developers have very limited support to build a correct payload.

AutoJMH provides the automatic generation of payloads, starting from a code segment selected in a large application. Our generative technique prevents two of the most common mistakes made in microbenchmarks: dead code elimination and constant folding. A microbenchmark is such a small program that it can be "over-optimized" by the JIT and result in distorted time measures, if not designed carefully. Our technique automatically extracts the segment into a compilable payload and generates additional code to prevent the risks of "over-optimization". The whole approach is embedded in a tool called AutoJMH, which generates payloads for JMH scaffolds.

The capabilities of AutoJMH were validated, showing that the tool is able to process a large percentage of segments in real programs. We also show that AutoJMH can match the quality of payloads that are handwritten by performance experts and outperform those written by professional Java developers without experience in microbenchmarking.

1.2.3 Lossy Compression of Programs

Image and sound are approximate by nature, while source code and assembler instructions have been traditionally considered to be strictly exact data. Recent research reports two phenomena that challenges the idea of code as exact data: Forgiving Zones [Mit16, XMK16] and Program Diversity [BM15, BAM14]. Forgiving zones are areas of the program that can endure some level of approximation and still produce good results. Program diversity occurs when two programs that were implemented in two distinct ways, seem to be the same to an observer. Like two different images perceived as the same by a user, if program A can produce the same results as program B, we can say that A is an approximation of B. This example proves that the areas where Approximate Computing can be used are far from being exhausted.

Based on the idea that programs are in fact approximate data, this thesis introduces Primer, the only (to the best of our knowledge) lossy compression algorithm for ARM assembler instructions. Primer exploit forgiving zones and diversity to compress programs in a lossy matter: the decompressed program might contain some degree of degradation in the form of bit flips on its assembler instructions and still complete its execution correctly. Our experiments show that by losing information, Primer is able to improve on the state-of-the-art code compression methods by a 10%.

1.3 Outline

This thesis is organized as follows.

Chapter 2 presents the state of the art. This chapter surveys existing techniques to identify forgiving zones, enumerate a large part of the existing techniques for approximation and then it goes into details on how the resource gains and accuracy losses are measured.

Chapter 3 presents the first contribution of this thesis, Approximate Unrolling. There we explain the syntax and semantics of the loops targeted by the optimization. Its implementation on the OpenJDK compiler is also described in detail and finally, a thorough evaluation is presented, where the execution times and energy consumption reductions and accuracy losses obtained as results of using the optimization are evaluated.

Chapter 4 illustrates AutoJMH. Initially, a introduction on microbenchmarking (as well as the challenges to it) is presented. The chapter divides microbenchmarking into scaffolding and payload. The scaffolding part is constructed using previous works, while a novel method for generating payloads is described. The chapter also shows the experiments conducted to evaluate AutoJMH, where it was shown that the tool outperformed Java Engineers without experience in microbenchmarking.

Chapter 5 describers Primer. Initially, the chapter illustrates in-depth the foundations for a code recovery mechanisms that allows Primer to compress the ARM instructions. Secondly, we explain the basis for the lossy part of the algorithm, which exploits forgiving zones to compress the information a lossy way. The chapter finishes showing the evaluation of the tool, which is able to improve by 10% the current state of the art algorithms for code compression.

Chapter 6 concludes the thesis, providing a summary of its contributions and outlining our future work in Approximate Computing.

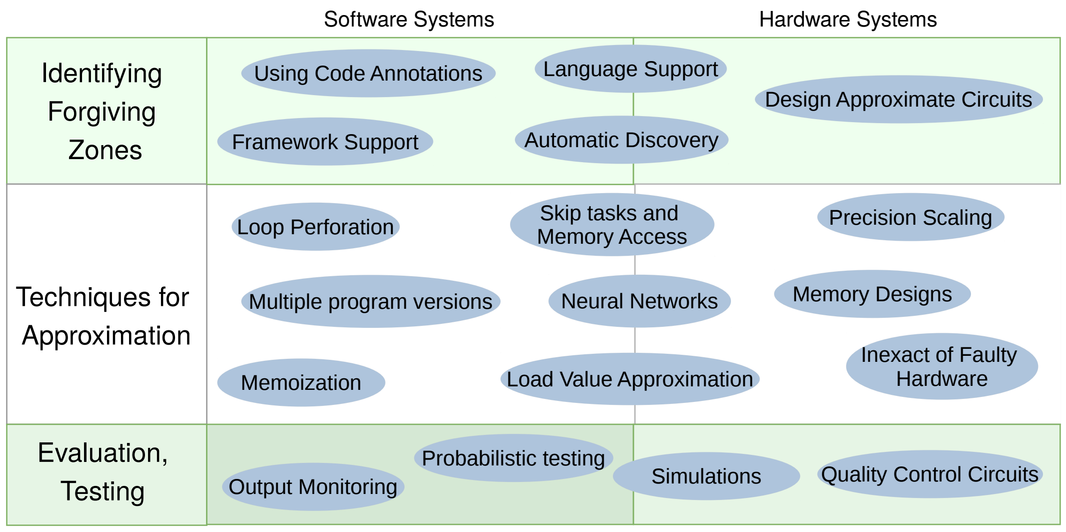
Chapter 2

State of the Art

Employing the imprecise nature of data to obtain CPU time or storage space is not a new idea. Approximation has been exploited, pretty much ad-hoc, for quite some time now. In some fields, approximation is natural (video, sound) or even inevitable (floating point operations). Traditionally, it has been considered a factor to deal with.

The novelty of Approximate Computing lies in the mental shift that goes from using only those imprecisions that came naturally (or avoiding it altogether), into actively seeking new approximation opportunities. Practitioners of the field consider accuracy not a goal, but a resource. Hence, approximation becomes an opportunity rather than a problem.

Figure 2.1: Approximate Computing Field map. Each bubble represent a technique described in this chapter. Rows divide techniques by the challenge being addressed, while columns indicate whether the technique is designed for Software or Hardware.



This chapter provides a survey of the Approximate Computing's field. It begins by mentioning other surveys in the field that can help the reader widen their perception of the state of the art. Afterwards, the chapter details challenges faced by Approximate Computing practitioners, such as defining a Quality of Service metric, identifying Forgiving Zones and measuring accuracy losses and resource gains.

Addressing Approximate Computing' challenges are the driving factor for most researchers in the field and clearly divide the area in research directions, providing initial classification criteria. The other two criteria used in the chapter are (a) software/hardware techniques and (b) its field of application. These criteria allow to build a field's map like the one shown in Figure 2.1, which in turn serves as guidance to describe the numerous works in the area.

This chapter ends by discussing how each individual contribution advances Approximate Computing as a whole.

2.1 Related Surveys

Other authors have proposed surveys on Approximate Computing. The most up to date and complete one is authored by Mittal [Mit16]. This review shows the many strategies for identifying forgiving zones, mentions a large group of techniques for approximation and enumerates the hardware components targeted by each technique.

While some works in the area are software-only, the bulk of strategies for Approximate Computing exploits opportunities in hardware. Hence, another survey worth of mention is the one by Xu et al. [XMK16]. This review, while being less exhaustive than Mittal's, provide a much more in-depth vision of hardware techniques. This focus is also shared by the survey of Han et al. [HO13] which also gives special attention to hardware.

The body of work related to Approximate Computing is considerable. Some areas are large enough to deserve surveys of their own. Jian presented a comparative study on approximate arithmetic units [JLL+17] and then another on approximate multipliers [JLM+16], while Dutt [DNT16] proposed a survey on approximate adders only.

Outside academic publications, Sampson maintains a web on-line survey [Sam17] adding entries as the field progresses. While not quite in-depth, this website is the field's most exhaustive survey, freed from the space restrictions imposed to publications.

Contributions of this Review Besides the required mention of the techniques to first identify and then approximate forgiving zones, this chapter contributes to the existing reviews a detailed enumeration of the strategies used to evaluate the accuracy losses and resource gains, which the previous surveys tend to somewhat overlook. The importance of this is that while the works on Approximate Computing presents remarkable savings in performance and energy by relaxing accuracy requirements, the way these gains are measured remains in the background. As example, energy consumption in nano-scale circuits is obtained using models [YFE+07] and simulations [GMP+11], while performance can be notoriously difficult to measure properly [Ale14a]. Understanding how results are obtained provides a novel and complementary point of view on the field that has not been explored before.

2.2 Challenges to Practitioners

Current results in the area are very promising. Yet, effectively using accuracy as a resource is by no means an easy task. This requires practitioners to (i) use existing domain-specific knowledge to choose a Quality of Service metric to rule whether the system still produce acceptable results, (ii) identify the system's forgiving zones, (iii) select or design suitable approximation strategies and finally, (iv) evaluate both the functional (correctness, safety) and non-functional (performance, energy consumption) system' requirements. While this chapter's novelty lies in the enumeration and analysis of existing measurement techniques, we address all challenges for the sake of completeness.

Challenge #1: Selecting a proper Quality of Service Metric QoS metrics encode the domain-specific knowledge needed to determine if a technique maintains the system within tolerable accuracy boundaries. Being domain-specific, there is no one-size-fits-all QoS. Instead, a specific QoS must be selected for each application. For example, works that apply approximation in the area of Signal Processing often use the Signal to Noise Ratio (SNR) of the produced signal as QoS [SLJ$^+$13, RRWW14], while works on Machine Learning uses classification errors [MRCB10]. A proper selection of the QoS is of vital importance, since it plays an important role in identifying forgiving zones and evaluating the approximate system [CCRR13].

Challenge #2: Identifying Forgiving Zones. The fact that not every part of an artifact can be approximated has been repeatedly observed by individual researchers in the area, which have reported 'approximable' [ESCB12], 'relaxable' [YMT$^+$15] and 'forgiving' [VCC$^+$13] zones, as opposed to 'strict' [YMT$^+$15] 'sensitive' [CCRR13] or 'precise' [SDF$^+$11b] parts.

Identifying parts amenable for approximation is required since most approximate techniques are unable to discover such amenable parts without guidance. Most approximate techniques are unsound in nature and require some human guidance or dynamic analysis to ensure correctness. For example, a situation in which Loop Perforation [SDMHR11] works best, is loops improving an already good value through successive iterations. Even assuming that is possible to detect such loops, the 'good' value cannot be known simply by analyzing the code. Some extra mechanism is needed from the programmer (a QoS function or annotations) to specify or detect such values. Another example: in the Parapprox's [SJLM14] stencil pattern, energy savings are obtained by skipping memory addresses in images, using neighboring values instead. This only works if neighboring pixels are indeed similar. Is possible to dynamically determine that this condition holds by accessing those skipped pixels. However, this will obviously nullify the savings, requiring human analysis.

These examples prove that mechanisms to detect approximate parts of a system are needed, whether they are automatic analyses or manual annotations. Section 2.3 describe the works providing such detection instruments.

Challenge #3: Provide approximation strategies for the system After Forgiving Zones have been detected, a number of approximation techniques can be used to trade-off accuracy for performance, storage or energy.

There seems to be no universal approximate technique that can be always applied with good results. Some techniques are designed for hardware systems [SDMHR11, BC10, VPC$^+$15], while others for software [GMP$^+$11, JLM$^+$16, VRRR14]. Also, not all techniques trade accuracy for the same purpose. Most importantly, each approximate technique has known situations and domains in which it works best. Using an approximate technique without a proper analysis of whether it will fit a particular scenario, can have null or even adverse results such as loss of performance or unacceptable levels of imprecision. A careful study of available techniques is required to determine those providing the best result for a particular application.

This chapter provides an extensive survey of existing techniques for approximation in Section 2.4.

Challenge #4: Evaluating an Approximate System As mentioned, Approximate Computing has the opposing goals of improving on some resource efficiency at the expense of accuracy while maintaining imprecision to an acceptable level. These two objectives constantly in tension adds complexity to the evaluation and testing of an approximate system. Approximate systems must be not only proven correct, but also must be evaluated w.r.t reductions in some resource consumption to justify the loss in accuracy. The nature of some approximate techniques creates stochastic or probabilistic systems [SPM$^+$14] forcing designers to come up with new ways of testing. Also, when legacy systems are being made approximate, it can occur that the existing test suite is not designed with approximation in mind, which forces a redesign of the system's test [BARCM15b].

2.3 Identifying Forgiving Zones

This section presents several approaches to identify forgiving zones in a system. Previous research has presented frameworks [BC10], language support [SDF$^+$11b, MCA$^+$14, PEZ$^+$15] or annotations [VPC$^+$15] to manually specify forgiving zones in a program. Some other techniques [RRWW14] propose an automated process, relieving the programmer from analyzing the system. On hardware, Forgiving Zones are found using Quality Control Circuits [LEN$^+$11, RRV$^+$14, VSK$^+$12]

This section tries to prioritize those works that contribute mainly to identify forgiving zones. Notice that early papers describing approximate techniques such as [SDMHR11, RGNN$^+$13] usually proposed some way of detecting the forgiving zone they addressed as well. This was done precisely due to the lack of work explicitly targeting this challenge.

2.3.1 Automated Approaches

Alter and Test. The general idea behind most fully automated approaches to discover forgiving zones, is to alter all potentially forgiving zones of the system in some way. Then, several tests are performed on the modified system to see if the QoS criteria are met If so, the zone is marked as forgiving. The output of these automatic tools is a group of forgiving zones existing in the program.

In [RRWW14] variable values are randomly altered to discover which variables have little impact in the program's QoS and therefore can be approximated. The ARC [CCRR13] framework also modifies variable values to discover resilient code segments and then tries several approximation strategies to determine not only the existence of a forgiving zone, but also the best strategies to use in each zone. DynamicKnobs [HSC+11] discovers the system's input parameters having an impact in the QoS-performance trade-off using a training phase in which the program is run multiple times with different input values. A Quality of Service profiler is presented in [MSHR10] where Loop Perforation is used to approximate candidate zones and report QoS results back to the developer.

On Hardware Systems the alters and test methodology is also used. Probabilistic Pruning [LEN+11] is a method that models a circuit as a graph (wires are edges and nodes components). The method then iteratively removes edges and nodes. If the circuit does not meet the QoS after some node or edge is removed, the removal is reverted and another component is tested. The ASLAN framework replaces exact adders and multipliers in sequential circuits by approximate ones and then uses a Quality Evaluation Circuit to determine if such replacement affects the QoS to acceptable levels. A similar idea is presented in other works as well [VSK+12].

Interestingly enough, the alter and test approach is also used to find forgiving zones for purposes other than (but highly related to) approximate computing, such as error-resilience and diversity [BAM14]

Static Analysis. A number of works in the area of fault tolerance uses static analysis to detect critical regions of code that must be protected in order to ensure correctness. These approaches are interesting for Approximate Computing practitioners, since those results can be used to narrow down the search for forgiving zones.

Critical Instruction Analysis and Protection [CG11] is a method that uses static analysis to separate critical from non-critical operations. Related to this is `SJava`, a language that can be used to discover errors that only disrupt the program for a limited execution time. In [HMS+12] a methodology is proposed that maps circuits to higher abstractions allowing to perform static analysis. This is done to distinguish critical components such as those controlling the execution flow or handling memory addresses from others that can be approximate.

2.3.2 Manually Identifying Forgiving Zones

While automated approaches can discover forgiving zones without human intervention, they require lengthy trial and error training phases and the forgiving zones discovered are usually limited to those resilient to a particular set of approximation techniques. Such tools are also unable to detect all zones identified by humans (for example, ASAC [RRWW14] detects 86%). Sometimes the lengthy training phases can be avoided since a forgiving zone can be self-evident for an expert or the zone is omitted by the tool, therefore a way to allow human intervention to identify forgiving zones is needed.

Fortunately, there is quite some work addressing this issue. Several languages to work with approximation have been proposed [SDF+11b]. Others authors propose an OpenMP-style of annotations to work with approximation [VPC+15]. Another approach is to create types that expose the semantics of data approximate in nature [BMM14, SMR15].

Language Support. Approximate Languages are languages that allow the programmer to deal with approximation. Several approximate languages rely on adding support to existing languages in the form of annotations to identify forgiving zones [CZSL12, ENN13]. EnerJ [SDF+11b] allows to annotate variables in Java program whose values can be approximate. The same approach is followed by Chisel [MCA+14] for C, also allowing to annotate operations and to specify the permitted approximation level. The authors of FlexJava [PEZ+15] claim that their language allows the programmer to write considerably fewer annotations than preceding approximate languages. In hardware, Axilog [YMT+15] is a set of language extensions for Verilog that provide approximate support for hardware design.

The Eon language [SKG+07] follows the Domain Specific Language (DSL) path, completely defining a new language. Eon allows programmers of embedded devices to explicitly define several equivalent 'flows' (i.e. sequences of operations) to execute depending on the varying levels of energy available to the device. Another accuracy aware DSL is SLax, that is used to define latency, loss and value-deviation tolerances of data acquired by sensors [SMR15]. The DSL is used in conjunction with LAX, a C/C++ API that allows reading sensor data values specifying the range of accuracy the caller can tolerate.

Precompiler Directives Rahimi et al. [RMGB13] propose a multicore cluster containing floating point units (FPU) allowing several levels of accuracy. They used OpenMP-like extensions to identify the approximable parts of a program that could be executed using lower levels of accuracy in the accuracy-configurable FPU units of their architecture. Precompiler directives are also used by Vassiliadis et al. [VPC+15] to allow programmers express the impact of computations on the QoS and to specify several alternative implementations to the same computation with different levels of accuracy and energy consumption.

Explicit Approximate API Another approach to identify forgiving zones is to use data types having a known approximate nature. `Uncertain<T>` [BMM14] is a .NET type for uncertain data supporting various non-deterministic operations. The type is used to express that some value exists in a range with a given probabilistic distribution. Another example is CES, an extension to C/C++ [Thr00] that introduces probabilistic types such as `prob<int>`. LAX [SMR15] is a C API to obtain readings from sensors. It exposes the approximate nature of sensors in the sense that readings are explicitly known to be approximate.

2.4 Strategies for approximation

Approximation techniques exploit opportunities in both hardware and software to trade-off accuracy for energy or performance. Works in the field propose to reduce computation precision, drop tasks or memory accesses in software and exploit inexact/faulty hardware. This section enumerates the many techniques for approximation existing today.

2.4.1 Hardware Techniques

Approximate techniques oriented to hardware employ accuracy-configurable floating point units, approximate adders, and faulty memory banks. Also, authors have proposed to automatically synthesize approximate circuits.

Precision Scaling Precision scaling techniques employ the numerical requirements of some applications to reduce the number of bits needed to represent values. Using fewer bits yields fewer active components in circuits, hence reducing energy consumption. In [TNR00] a number of ways to reduce bitwidth in floating point (FP) operations for healthcare applications are presented, resulting in no QoS losses. A similar idea is presented in [YFE$^+$07] for physics simulations with similar results.

Integer Precision Scaling has been also explored, Brooks [BM99] achieved up to 50% reduction in energy by noticing that benchmarks in SPECint95 and MediaBench rarely used more than 16 bits for integer operations.

ApproxMA [TZW$^+$15] is an approximate memory access framework that uses precision scaling to save energy, by means of reducing communications between chips and surrounding off-chip components, showing that the energy required to fetch data from off-chip memory is higher than that needed to perform on-chip computations.

Approximate Functional Units A considerable number of papers have proposed faster/energy efficient approximate adders and multipliers, becoming an area worthy of surveys of its own [DNT16, JLM$^+$16, JLL$^+$17]. Here we present some representative works.

IMPACT [GMP$^+$11] presents an adder that consumes only the 60% of its precise counterpart, having only one-third of the size. This is achieved by removing transistors from the adder cells. As result, the adder produces incorrect results in a small number of cases. In [DVM12] an approximate adder is proposed based on the observation that carry chains in random additions are always shorter than the total number of bits being added. Hence, the addends are divided into segments having a high probability of being longer than the carry chain. The segments are then added in parallel without any carry bit, since there is a high probability that no carry bit between sub-adders is needed. This increases the operation speed by means of parallelization.

Accuracy-configurable adders [YWY$^+$13, SAHH15] also work by subdividing the addends into segments and adding them in parallel using sub-adders. The difference is that consecutive sub-adders are connected by multiplexers that may or may not allow to pass the carry bit. Turning these multiplexers on/off the accuracy level can be tunned, in contrast with static adders. If a multiplexer is configured to forbid passing the carry bit to the next sub-adder, the adder uses some prediction function, which could be a constant $\{0,1\}$ or ands with the first bit from the previous segment.

Other approximate functional units such as multipliers have been proposed [JLM+16]. These designs propose strategies such as using approximate adders and truncate or approximate intermediate results in the multiplication.

Automatic Generation of Approximate Circuits The techniques presented so far propose circuit designs that have been manually designed. Several authors [LEN+11, RRV+14, VSK+12] have proposed synthesizing/modification techniques that automatically generate or modify circuits to exploit the accuracy trade-off. These techniques start with an accurate circuit that iteratively approximated while some QoS criteria are met Resulting from the transformations, reductions in energy consumption and circuit size are obtained.

Examples of these works are the previously mentioned Probabilistic Pruning [LEN+11] and ASLAN [RRV+14]. The former automatically approximate a circuit design by removing components and connections, while the later exchanges precise units by approximate ones. SALSA [VSK+12] proposes a general framework to approximate circuits by exploiting the existence of 'Don't Cares' i.e. bits of the output that do not care for some bits in the input. Miao et al. [MGO14] propose an Approximate Logic Synthesis framework that automatically generates approximate logic units ensured to deviates from precise answers only by a design-specified margin and where wrong results are produced with a frequency below a specified value.

Memory Design A special kind of circuits, memory have received quite some attention from the Approximate Computing community. The common driver for all approximate techniques addressing memory designs is to reduce the amount of energy required to store or retrieve data [Sam17]. Chen et al. [CYQ+16] proposes a multilevel accuracy memory circuit which is composed of several banks with varying degrees of accuracy. They present a data-significance circuit that matches data and banks depending on the significance of the data (i.e. the more significant data are stored in the most accurate banks). This approach can result in up to 60% savings. A similar idea is proposed by [RSJR17] to propose an accuracy-tunable system for DRAM banks, which needs to be refreshed frequently to keep data integrity and each refresh cost energy. This approach also divided the memory in banks with different levels of accuracy. By tunning the refresh rate of a DRAM bank, its accuracy-energy trade-off can be configured as less accurate banks are refreshed less frequently, hence requiring less energy.

In storage units that can wear out such as Flash or PCM memories, approximate techniques can increase the chip's lifetime. Sampson et al. [SNSC13] proposes to store approximate data in faulty memory banks that have exhausted their error correction budget, preserving those who have not. Their simulations indicate that this would improve memory lifetimes by 23%. SoftPCM [FLL12] increases the life expectancy of PCM memories allowing only those write operations in which the old data being overridden is dissimilar from the new data being stored. Their simulations indicate that around 22% of the write operations can be avoided.

Load Value Approximation Other authors have proposed to reduce the memory-wall (i.e. the difference between the CPU and memory speeds) approximating the data obtained from memory accesses rather than performing the full access.

Rollback-Free Value Prediction (RFVP) [YTE$^+$16] is a technique that predicts approximate values that miss in the cache. RFVP avoids checking or retrieving (i.e. drops) from memory a number of such approximate missed values. The number of dropped values can be adjusted to tune the accuracy trade-off. San Miguel et al. [MBEJ14] proposes a load value approximation technique that exploits value locality (the fact that in many application values stored nearby are similar) to predict cache misses. Contrary to RFVP, the actual value is indeed fetched from the next level, however, the CPU uses the approximate value, avoiding the stall. The accurate value is then used to train the predictor. Dissimilar values decrease prediction confidence. When confidence goes too low, values stop being predicted.

Doppelgänger [MAMJ15] is a cache design that groups similar values into a single one, reducing the amount of data stored in the cache. The authors report up to 2.5x improvements in energy efficiency.

Hardware-Accelerated Neural Networks Neural Networks (NNs) are highly parallel and can be accelerated by specific hardware (called Neural Processing Units), providing a general-purpose, superior-performance way of doing computations. Esmaeilzadeh et al. presented in [ESCB12] an approach where a NN learned the behavior of program segments. Later on, a NPU is designed to hardware-accelerate such network and the compiler is modified to replace the code being approximated by calls to the NPU. This resulted in performance and energy efficiency improvements. In [ERZJ14] is proposed the usage of hardware NNs implemented in Verilog[1] to replace a small set of the GNU C library (`glibc`) functions, i.e. `sin`, `cos`, `log`. They reported considerable improvement in energy consumption and performance. Other authors have proposed similar methods, also with good results.

Inspired by the success of previous works to approximate programs using NNs, some authors have proposed an even more aggressive method, which is to approximate the NN itself. In [VRRR14] a method is proposed that identifies the network nodes influencing the classification the least and selectively approximate those by implementing them in less accurate but cost-effective hardware. Similarly to this, [ZPL14] proposes to approximate an NN replacing exact arithmetic units by inexact ones. The accuracy levels of the newly obtained approximate NN are increased by repeating the original training phase (retraining). As result, energy savings between a 40% and a 60% are obtained with acceptable losses in precision.

[1]`http://www.verilog.com/`

2.4.2 Software Techniques

Approximate techniques in software automatically modify algorithms to drop task or memory accesses, propose unsound parallelization or provide multiple versions of the same program. This section expands on these ideas.

Dropping tasks and memory access Loop Perforation [SDMHR11] is a technique that skips iterations in loops. This idea can be successfully applied in loops iteratively improving an already good result, searching/ filtering values or finding the median for an array of numbers with uniform distribution, among others. Vassiladis [VPC^{+}15] proposes a general programming model and a runtime to selectively drop tasks in a program.

ApproxHadoop [GBNN15] proposes to approximate the Map-reduce framework by sampling (i.e., skipping some) input values and dropping tasks. Parapprox [SJLM14] is a framework that presents several ideas specifically tailored to common patterns in parallel such as using memoization to map atomic functions with similar parameters, skipping nearby memory accesses in video or sound applications since the values are likely to be similar or reducing the parameter number passed to a function by taking only those affecting most the output and providing default values for the rest.

Software Memoization Memoization has been also used in software-only systems to approximate hardware. The general idea is also to return the cached value of similar computations instead of performing the complete computation. Agosta et al. [ABCF11] proposed a software memoization framework for financial applications, while Parapprox uses approximate memoization to approximate the map pattern.

Synchronization Elision A series of papers have explored the idea of reducing synchronization between parallel threads to speed up computations. Synchronization Elision can improve performance by means of reducing wait operations and off-chip memory accesses.

Meng et al. [MRCB10] proposes to drop task causing large off-chip traffic both in terms of data accesses and coherence, but having little effect on the accuracy of the task being performed. The authors report speedups ranging from 4X to 8X using this technique reducing the QoS to acceptable levels.

HOGWILD! [NRRW11] proposes a parallelization scheme for Stochastic Gradient Descent algorithms that reduces lock rate and increases performance by allowing multiple processors to share memory. This introduces the possibility of two such processors destroying each other's works. However, in the cases when this is infrequent, the authors show considerable performance gains up to 7X.

Renganarayana et al. [RSNP12] propose a general framework to relax synchronization in programs by allowing programmers to specify non-synchronizing versions of functions. In their experimental dataset, speedups of up to 13X are obtained without noticeable QoS reduction.

Using multiple approximate versions Several works have proposed ways to allow the programmer to supply multiple approximate versions of the same program. PetraBricks [ACW$^+$09] presents a language and compiler that emphasizes in the ability to select between multiple algorithms for one same task. Beside algorithmic choice, the language features different levels of choice granularity, automatic consistency checks and specifying several executive orders.

Baek proposes Green [BC10], a framework that allows a programmer to specify multiple approximate implementations. The framework produces a QoS model from several training runs, exploring the different combinations of approximate implementations. Then, this model is used to dynamically select from different implementations upon varying requirements of energy consumption and performance.

Venkataramani [VRLS15] proposes to match the complexity of Machine Learning classifiers to the simplicity of the problem at hand. In other words, simpler problems should be solved using less complex classifiers (which should be faster and less energy hungry) leaving the more sophisticated for harder problems only.

ViRUS [WS14] propose to select from multiple virtual functions to match energy and performance requirements at runtime. SAGE [SLJ$^+$13] provides a framework for graphics engines that combines a compiler that automatically generates several kernels with varying values of accuracy/performance. The framework also provides a runtime able to select between those kernels when the performance requirements vary.

Lossy Compression Formats Lossy compression formats are among the most straightforward version of approximate computing. They exist long before the coining of the 'Approximate Computing' term. Perhaps that is the reason most surveys in the field do not include them. Industry standard such as JPEG [Wil93], AAC[2] and MPG[3] are well known file formats exploiting approximation to reduce storage space requirements.

Approximating Compilers Approximating Compilers exploits unsound (approximate) techniques to improve some aspects of a program, just like Optimizing Compilers optimize a program using sound (exact) transformations. Examples of already mentioned works proposing approximate compilers are PetraBrik and SAGE. Other works [SDMHR11, SDF$^+$11b, MCA$^+$14] modify or propose a compiler to demonstrate the feasibility of a technique by experimentation.

ACCEPT [SBR$^+$15], deserves special attention, as it is the compiler having the largest toolbox of approximate techniques. ACCEPT features Loop Perforation, Neural Acceleration, Synchronization Elision and Approximate Strength Reduction, which replaces arithmetic operations with lest costly shifts and masks that may not produce exactly the same result.

[2]`https://en.wikipedia.org/wiki/Advanced_Audio_Coding`
[3]`https://en.wikipedia.org/wiki/MPEG-1`

2.5 Evaluation of an approximate system

Properly evaluating an Approximate System, demands an assessment of both functional and non-functional requirements. Just like any other system, an approximate system is expected to successfully perform a series of task. However, since the system has been approximated, there is an expected gain in some resource resulting from relaxing the correctness requirements, which must be also measured to justify the accuracy losses.

Not every work in approximate computing trades accuracy for the same purposes. Table 2.1 resumes the expected gain of the proposed techniques of this survey. The most common targets are energy efficiency and performance. However, other resources such as storage space, circuit area and memory lifetime are also targeted.

We now provide some background on how approximate systems are evaluated. We describe the tools used for this purpose, the QoS metrics employed, the different resources gains for which accuracy is relaxed and how such gains are measured.

Table 2.1: Different resource gains expected of the works presented in this survey

Expected Gain	**How is Measured or Estimated**	**Example Works**
Performance Speedups	Simulations	Approx. Load Value [TPY+14], NPUs [ESCB12]
	Microbenchmarks	Approx. Adders [DVM12]
	Benchmarks	Tasks Drops [SDMHR11]
	Wall Clock	Task Drops [GBNN15], Sync. Elision [MRCB10]
Energy Savings	Simulations	Approx. Adders [GMP+11] Approx. Synthesis [LEN+11, RRV+14], NPUs [ERZJ14]
	Energy Models	Precision Scaling [YFE+07, TZW+15], NPUs [ESCB12, VRRR14]
	Measurements	Memory Design [RSJR17]
Storage Space	File size	that
Memory lifetime	Simulations	Solid State Memory Design [SNSC13, FLL12]
Circuit area	Estimations	Precision Scaling [YFE+07], Approx. Synthesis [RRV+14]
	Measurements	Approx. Addders [GMP+11]

2.5.1 QoS metrics

As mentioned, QoS metrics play a central role in approximate systems. They encode all domain knowledge needed to determine where and to which extent the system can be approximated. Automatic detection of forgiving zones uses the metric to detect the

Table 2.2: Some of the most frequent QoS metrics used to evaluate Approximate Computing techniques.

QoS	**Workload Type**	**Example Works**
Error Rate	Monte-Carlo Simulations, Speech Recognition	Precision Scaling [TNR00], Approx. Arithmetic Units [SAHH15, DVM12]
Error/Relative Error Distance	Image Processing	Approx. Synthesis [RRV$^+$14] ,
Dice Metric	Img. Processing	Precision Scaling [DVM12]
Normalized Root Mean Square Error	SPEC CPU 2000/2006	Approx. Load Value [TPY$^+$14]
PSNR	Signal Processing (Image, Video, Sound), Games	Task Drop [SJLM14] Approx. Adders [GMP$^+$11], Synthesis [RRV$^+$14], Memory Designs [RSJR17], Multiple program versions
Clustering Errors	k-means and k-nearest clustering,	Precision Scaling [TZW$^+$15], Synthesis [RRV$^+$14]
Correct/Incorrect Classifications	ML Object Recognition	Memory Designs [RSJR17], NPUs [VRRR14], Sync. Elision [MRCB10]
Ranking Accuracy	Document Search, Supervised Semantic Indexing k-means	Multiple program versions [BC10], Sync. Elision [MRCB10]

approximable system parts. In techniques where accuracy can be dynamically adjusted, the QoS is used as a limit indicating where accuracy losses cannot be tolerated any further.

Table 2.2 shows some of the QoS metric used by the surveyed authors. Column ‘QoS’ shows the QoS metric used, column ‘Workload Type’ the type of workloads on which this QoS metric is used and ‘Example Works’ shows some of the works that used this QoS metric to evaluate the proposed technique.

The list of metrics in the table is not exhaustive, yet it illustrates some of common QoS metrics found in our survey. Among those metrics are the Error Rate, which is the probability of an operation yielding an incorrect value. Error Distance is the absolute distance between precise P values and approximate ones A ($ED = |P - A|$). The relative Distance is the Error Distance divided by the precise value $RD = ED/P$. The Dice metric returns the ratio of different elements in two sets (i.e. pixels in images). The Signal To Noise Ratio (SNR) is a metric to measure the level of noise in a given signal. The Ranking Accuracy is a metric that compares two lists. It highly rewards elements with the same index in both lists. Lesser rewards are given to elements with different orders in both lists. Elements not in both lists are penalized.

2.5.2 Evaluating Accuracy Losses

Most authors perform accuracy loss measurements simply by comparing the approximate system's output against its precise counterpart using the QoS metric. Nevertheless, a relevant body of work targets building tools specifically to evaluate the QoS of approximate systems, such as profilers [MSHR10], debuggers [RSA$^+$15], assertion frameworks [SPM$^+$14] and monitors [GR14]. This section illustrates some of the existing tools specially meant to evaluate accuracy loss.

Off-line Tools Misailovic [MSHR10] proposed a Quality of Service profiler. The profiler transforms a group of loops using the Loop Perforation [SDMHR11] technique and reports the QoS loses resulting from such transformations to the programmer. While the profiler is limited to the usage of a single technique, it would be possible to extend the idea to other approximation strategies as well.

Ringenburg [RSA$^+$15] proposed a QoS debugger which works by taking into account number of times a particular operation is executed as compared to the number of times it produces an incorrect result. It also observes a number of approximate operations that converge in a particular segment to highlight zones heavily affected by accuracy.

A probabilistic assertion framework was proposed in [SPM$^+$14], allowing developers to write assertions that test the probability of a value existing in a given range with a given confidence. The framework is meant to test programs running on inaccurate or faulty hardware for which the functional requirements must be relaxed.

On-line tools Grigorian and Reinman [GR14] propose Light-Weight Checks (LWC) based on the observation that while obtaining a solution could be difficult, checking its quality could be simple. A LWC inspects the computation's outputs against a user-provided QoS function and reruns those computations for which the QoS is not meet

Ringenburg et al. [RSA$^+$15] also introduced a QoS monitor. A monitor is an on-line tool meant to be used while the system is in production to automatically adjust accuracy parameters. The monitor compares results of a precise operation sample set with their approximate counterparts. At runtime, the system administrator could control the frequency of those checks to reduce their impact on performance. Verification was performed using programmer-provided verification functions and Fuzzy Memoization [ACV05] that compares recorded values from similar operations to those obtained by the approximate ones.

Rumba [KZSM15] is a framework that proposes to monitor the output of hardware accelerators such as the one for Esmailzadeh [ENN13] using LWCs. When the accelerator is being built, a hardware error model of the accelerator is created. The resulting model is then used to continuously monitor the accelerator's output. The novelty of their approach is that they propose to do so continuously, as opposed to others [RSA$^+$15, GR14] , which only do so for a subset of the computations.

Quality Control Circuits (QCCs) While no author in our survey discusses QCCs in isolation, they are worth mention. These circuits are hardware tools solely aimed at monitoring the QoS of an approximate hardware system and several authors [LEN+11, RRV+14, VSK+12] use them to control the QoS of their designs. The way QCCs works is by receiving digital signals from two different circuits and comparing them. If the difference is below a given threshold an 'accepted' signal is output.

2.5.3 Evaluating Performance Gains

This section provides a very condensed view of Performance Engineering methodologies and tools with a focus on their usage by an Approximate Computing practitioner.

Profilers Profiling is a well-established technique having commercial tools such as YourKit[4] and JProfiler,[5] which allows to observe programs as a whole and determine the program parts where most of the execution time is spent. While not used to measure performance gains per-se, profilers have found use in approximate computing to detect system parts where most of the execution time is spent.

As example, Chippa [CCRR13] uses profilers to determine the runtime fraction used by the forgiving zones of the benchmarks used for experimentation, while Sidiroglou [SDMHR11] uses profilers to detect the program loops responsible for at least 1% of the execution time and then performs experiments over this subset

Benchmarking Performance Benchmarking is a measurement technique that executes a computational intensively workload with a given set of inputs and then measures the time it requires to execute. Commercially available benchmarks such as SPEC [6], SciMark [7], PARSEC [8] and others, are maintained by a team of dedicated experts.

This measurement strategy is frequently used by Approximate Computing practitioners since it is primarily meant for comparison purposes. Also, benchmarks are publicly available and are constantly revised and maintained, allowing to reproduce and compare results easily. Depending on their technique's design, authors use SPEC [SNSC13, RRWW14], SciMark [9] and PARSEC [SDMHR11, MSHR10] or a combination of them [SDF+11b, MSHR10]. Some others even use their own [MCA+14].

The benchmarks reviewed vary in size, some require repetitive executions to observe speedups, while other (mostly in large-scale data [GBNN15] and document search/indexing [BC10] applications) have workloads with execution times large enough to be measured in seconds. The execution times in these systems are large enough that they can be measured in one run simply by using the system timers.

Microbenchmarks Microbenchmarking is related to Benchmarking. This measurement methodology is used when the workload's execution is extremely short (nano and

[4] `www.yourkit.com/`
[5] `www.ej-technologies.com/products/jprofiler`
[6] `www.spec.org`
[7] `math.nist.gov/scimark2/`
[8] `parsec.cs.princeton.edu`
[9] `http://math.nist.gov/scimark2`

milliseconds lapses must be observed), as opposed to larger workloads in the previous paragraph. Microbenchmarking works by repeatedly running a tiny workload (a variable assignment, a method call) in isolation for a fixed amount of time. Through repetition, otherwise imperceivable delays are accumulated. Then, knowing the number of times the snippet was run, is possible to estimate, (with a given confidence margin) the time needed to run it once. Microbenchmark's adoption is somehow hindered by the high level of expertise required to properly design a test and then interpret the obtained results [Ale14a]. Commercial tools for microbenchmarkings are JMH[10] and Google Caliper.[11]

As example of microbenchmarking usage, Du [DVM12] used a C++ program running a Monte Carlo simulation with 10 million unsigned random inputs to evaluate their approximate adder.

Simulation Hardware systems use simulation extensively to measure performance gains. The advantage of simulation is that it allows faster cycles of research and development. Tools commonly used for performance simulation are MARSSx86 [12] and Synopsys PrimeTime VX [13]. As an example, Thwaites [TPY+14] uses the MARSSx86 simulator to measure the reduction in the memory wall. This tool is also used to measured the speedups obtained by using the Parrot transformation [ESCB12], which employs NPUs to gain performance. Rahimi [RMGB13] uses the PrimeTime VX simulator to do timing analysis on their shared FPU designs.

2.5.4 Evaluating Energy Consumption

Determining the energy consumption of a particular component is by no means an easy task. In many systems, energy consumption varies depending on many factors such as temperature and workload, so repeatability is challenged. It might also be the case that the power requirements of a component must be estimated since it is only possible to measure the power supply of the entire circuit (for example, estimating the consumption of nano-scale approximate adders in a CMOS circuit). In software, energy measurements can be even more difficult as power consumption is distributed around many components, CPU, memory, storage, etc. Also, in most modern computers there are many software systems running concurrently.

Simulation Energy consumption is frequently obtained using simulations in hardware systems. When the system under study is well known and understood, it can be simulated with a high degree of confidence. This provides a representation of the circuit's consumption that is less costly and faster to obtain than actual measurements. Known disadvantages of simulation are that it requires a detailed specification of the expected inputs and the activity of the circuit. Najm [Naj94] describes this as the

[10] `http://openjdk.java.net/projects/code-tools/jmh/`
[11] `https://github.com/google/caliper`
[12] `http://marss86.org/~marss86/index.php/Home`
[13] `https://www.synopsys.com/implementation-and-signoff/signoff/primetime.html`

pattern-oriented simulation problem. In other words, the simulations are only accurate for the pattern of circuit's inputs and operations being simulated and by no means for all possible operational states and inputs.

Authors in this survey [dP11, YMT$^+$15, RMGB13], use tools such as Synopsys Compilers [14], SystemC [15] and FloPoCo [dP11] to simulate energy consumption. Yazdanbakhsh et al. [YMT$^+$15] use the Synopsys Design Compiler to measure the energy consumption of the circuits developed using their Axilog language. Other authors use a combination of tools; in [RMGB13] the SystemC is used to model a set of FPU units whose register-transfer-level design is made using FloPoCo.

Power Models A Power Model estimates energy consumption as a function of an already known parameter, such as the components present in a circuit or the number of instructions executed by a program. Different Power Models are used by the authors surveyed in this chapter to estimate power consumption. Power Models can be part of simulations. Also, they are used as stand-alone tools to estimate power. These models have varying degrees of accuracy. Models can be highly accurate if the system is well understood or somewhat coarse when the system is not so well understood or simplified on purpose to ease the calculations.

Works in this survey utilize different Power Models. Yeh et al. [YFE$^+$07] uses the PowerTimer [BBS$^+$03] power model to estimate the energy consumption of approximate adders. PowerTimer consists of a set of functions derived from empirical experiments or pure analytical equations to estimate the power consumption of basic components or a hierarchy of them. [TZW$^+$15] Uses the CACTI power model [NNR09] to measure the energy savings of the ApproxMA approximate memory system. On software, several authors [VPC$^+$15, MCA$^+$14] use the number of CPU instructions executed by the programs in their benchmark to estimate power consumption. Sampson et al. [SDF$^+$11b] built their own model based on previous power estimations studies.

Measurements The most straightforward way of determining the power requirements is by using a consumption measurement tool. As mentioned before, this might be not always possible. In the reviewed literature in this survey, such measurements are made when the differences between precise and approximate systems are relatively large [GBNN15] or when the consumption of a whole macro component (such as a memory board) was being measured [RSJR17].

2.5.5 Evaluating Circuit Area & Memory Lifetime

Circuit Area Some works relax accuracy in exchange for circuit area [YFE$^+$07, GMP$^+$11, DVM12]. Gupta et al. [GMP$^+$11] actually built the circuit and performed measurements. Other works base their results on estimations. This is done since hard-

[14]`https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html`

[15]`http://www.accellera.org/downloads/standards/systemc`

ware is difficult and expensive to fabricate. Yeh et al. [YFE+07] use data from previous works to estimate circuit area .

Memory lifetime Expectancy Memory lifetime in solid-state memories is yet another resource traded for accuracy. This kind of memories can be written only a limited number of times, (i.e. have a limited lifetime). All authors in this surveys perform extensive simulations to estimate memory lifetime. Actually measuring lifetime will require to wear out physical circuits to perform experiments, therefore, simulation is the most reasonable alternative.

As an example, Sampson et al. [SNSC13] build a statistical lifetime model observing the number of writes to memory of the applications in their workload when fed with different inputs. On the other hand, to evaluate SoftPCM [FLL12], the authors use a fixed set of inputs and applications (video compression methods). There, instead of a statistical model, the authors use the exact number of writes performed by their workload to estimate the lifetime savings of the proposed technique.

2.6 Applications of Approximation

In this section we describe the domains of application of Approximate Computing. Currently, the techniques of the field are applied heavily in Machine Learning, Multimedia processing, Search Engines and others. Table 2.3 classifies the works surveyed in this chapter by the application domain their target

Table 2.3: Domain of Application of Approximate Computing

Application Domain	**Example Works**
Media (Sound, Image, Video) processing	Synthesis [RRV+14], Memory Design [RSJR17]
Object Recognition	Memory Design [RSJR17], NPUs [VRRR14], Sync. Elision [MRCB10]
Clustering algorithms (K-Mean, K-Nearest)	Precision Scaling [TZW+15], Synthesis [RRV+14], Memory Design [RSJR17]
Big Data Processing	Task Drop [GBNN15]
Document Search and Indexing	Sync. Elision [MRCB10]
Machine Learning	Multiple Implementations [VRLS15]

2.7 Advance the State of The Art

This section discusses how the contributions of this thesis advance the State of the Art. Initially, we discuss our vision for a universal approximating compiler. Secondly, we advance the types of data that can be considered approximate. Finally, we describe how to help lower the entry bar to Microbenchmarks.

2.7.1 Towards an Approximating Compiler

Implementing a compiler optimization is costly for every organization, both in financial and in human resource terms, so every optimization is carefully weighed before being included. Even so, commercial compilers are equipped with large optimization toolkits to maximize the number of situations they can optimize. According to [AJ09] the Hotspot compilers C1 and C2 are armed with nearly 75 different optimization strategies, while the GCC compiler website [GCC17] indicates that at least 78 distinct optimizations can be turned on/off with compilation parameters. In comparison, ACCEPT [SBR+15], the approximating compiler having the largest approximation toolkit is packed with a meager number of 6 approximate techniques.

While it is true that the number of optimizations included in an optimizing compiler gives only a rough, high-level idea of the number of situations the compiler can handle, it is certainly an illustrative comparison.

Advance the State of The Art In this thesis we propose a compiler-ready approximate optimization, broadening the set of cases that can be improved by an approximating compiler. Our optimization exploits data locality (a common situation in time series) to improves upon the existing state of the art by reducing accuracy losses and program crashes while still producing similar speedup gains.

2.7.2 Discovering new species of Approximate Data

The novelty of Approximate Computing lies in a mental shift that goes from dealing only with natural or unavoidable approximation to actively seeking approximation opportunities. Yet, most of the works in Approximate Computing today still address the so-called 'low hanging fruits' by targeting data for which their tolerance to approximation is well known. Mittal [Mit16] mentions this, saying that one of the challenges of the field is 'its limited applicability'.

Advance the State of the Art. We use the mental shift proposed by Approximate Computing to experiment with a type of data traditionally considered among the most exact ones: Assembler Instructions. This pushes the boundaries on data types considered amenable for approximation outside the traditional 'low hanging fruits', actively finding new opportunities and increasing the field's applicability as a whole.

2.7.3 Lowering the Nanosecond Entry Bar

Many authors [SNSC13, HSC+11] of software approximate computing perform speedup experiments using existing benchmarks. Then, the execution times for both benchmark programs (precise and approximate) are compared. This approach has advantages, is simple and universally accepted. Also, carefully crafted, publicly available benchmarks allow reproducing results better.

Unfortunately, many workloads are complete programs on their own. Therefore, the benchmarking method ends up measuring the forgiving zone's impact on the workload, rather than the optimization's impact on the forgiving zone. If a particular zone has little impact on the system's performance, the speedups provided by the approximation technique would go undetected, even if the optimization yielded high speedups on it. The opposite also holds, if the zone has a huge impact on the system's performance, large speedups will be reported in the benchmark test, even if little improvement is obtained on the forgiving zone itself.

The way researchers cope with this concern is to limit the applicability to zones abiding to some criteria, usually to the most performance-critical forgiving zones [SDMHR11]. This suffices to demonstrate the feasibility of the technique but also has drawbacks:

- It reduces applicability by forcing dynamic analyses to discover performance-critical zones (i.e. it forbids ahead-of-time compilation).
- Leaves unanswered the question of how much a technique impacts its target, as only the zones' impact on the system is measured.
- Forbids to observe the effect of small tweaks and improvements.

One might argue that there is no point in optimizations having no significant impacts on the system. This makes sense for engineers building a single system. Yet, developers working on compiler and libraries cannot afford to think this way (and indeed, they do not [Ale13b]), as their code will be used in a myriad of different applications.

Enters Microbenchmarking Microbenchmarking is a complementary technique to benchmarking that tests a zone in isolation. It allows assessing the optimization effect on the zone, independently of the system. We claim that Microbenchmarking is complementary (not necessarily better) to benchmarking as it answers a different set of questions. Microbenchmarking is not without disadvantages, the most notorious one being how difficult is to properly design a microbenchmark test [Cli10, Ale15, Jul14b].

Advance the State of the Art We advance the State of the Art presenting a tool to construct microbenchmarks automatically. This tool is able to avoid common pitfalls and errors common to inexperienced designers. This approach lowers the entry bar to microbenchmark usage, yielding a more comprehensive assessment of the approximate technique's impact on performance.

Chapter 3

Approximate Loop Unrolling

This chapter describes Approximate Unrolling, a novel optimization that uses the ideas of approximate computing to reduce execution times and energy consumption of loops.

The key motivation behind Approximate Unrolling relies on the following observation: data such as time series, sound, video and images are frequently represented as arrays where contiguous slots contain similar values. As a consequence of this neighboring similarity, computations producing this data are usually locally smooth functions. In other words, computations producing or modifying nearby array values representing these kinds of data frequently yield similar results. Our technique exploits this observation by searching for loops where functions are mapped to contiguous array slots in each iteration. If the function's computation is expensive, we substitute it by less costly interpolations of the values assigned to nearby array values. In exchange for this loss in accuracy, we obtain a faster and less energy consuming loop.

Approximate Unrolling combines static analysis and code transformations in order to improve performance in exchange for small accuracy losses. Static analysis has two objectives: first, determine if the loop has a structure that fits our transformations (counted loops mapping computations to consecutive arrays slots); second, a policy estimates if the transformation could actually provide some performance improvements. To transform the code we investigate two possible strategies. One that we call Nearest Neighbour (NN), which approximates the value of one array slot by copying the value of the preceding slot. The other strategy, called Linear Interpolation (LI) transforms the code so that the value of a slot is the average values of the previous and next slot.

We modify the OpenJDK C2 compiler's source code to include Approximate Unrolling. We do so to avoid phase ordering problems, reuse internal representations that we needed for our analysis and to obtain guarantees that Approximate Unrolling can actually improve the performance of a highly optimized code created by a production-ready compiler.

We experimented with this implementation using a carefully selected set of real-world Java libraries. The objectives of our experiments were (i) to learn whether Approximate Unrolling is able to provide a good performance vs. accuracy trade-off; (ii) to understand the situations where Approximate Unrolling is best applied; (iii) to learn the impact of substituting more or fewer operations with interpolations; (iv) to compare Approximate Unrolling with Loop Perforation [SDMHR11], the state of the art approximate computing technique for loop optimization.

Our results show that Approximate Unrolling is able to reduce the execution time and CPU energy consumption of the x86 code generated by 75% to 40% while keeping QoS to acceptable levels. We also learned the situations on which Approximate Unrolling works best. Compared with Loop Perforation, Approximate Unrolling preserved better accuracy in 76% of the cases and raised fewer fatal exceptions than Loop

Perforation (2 vs. 7).

The contributions of this chapter are:

- Approximate Unrolling, an approximate loop transformation
- An efficient implementation of Approximate Unrolling inside the OpenJDK Hotspot VM.
- An empirical assessment of the effectiveness of Approximate Unrolling to trade accuracy for time and energy savings.
- An empirical demonstration of Approximate Unrolling's novel contribution to the state of the art, as an effective complement to loop perforation

The rest of the chapter is organized as follows, in Section 3.1 we describe our optimization and its scope. We then detail our implementation in section 3.2, our evaluation in section 3.3 and discuss results in section 4.3. Section 4.4 describes how Approximate Unrolling compares with other approximate computing techniques and finally, section 3.5 concludes.

3.1 Approximate Unrolling

In this section we describe the main contribution of the chapter, which is the Approximate Unrolling loop optimization. Initially, we use the example of Listing 3.1 to illustrate the kinds of loops targeted by the optimization as well as the process for the approximate transformation.

In the second part of the section, we formally characterize these intuitions. Since we implemented the optimization in a Java compiler, the formalization and examples will focus on the Java language. However, Approximate Unrolling is not language-specific. It can be used in any language featuring counted loops, statements, array assignments and boolean operations. The grammar and semantic rules given in this chapter can be adapted with little effort to a large family of widely used languages containing these constructs, such as C#, C++ or Python.

3.1.1 Illustrating the Approximation Process

Approximate Unrolling works by replaces the unrolled iteration's original instructions by others that interpolate results. We perform this transformation in such way that error does not accumulate as the loop runs. We currently implement two interpolation strategies: Linear Interpolation (LI) and Nearest Neighbor (NN). A policy determines which strategy to use in each loop.

Algorithm 1 summarizes the approximation process. It takes as input a program P, a loop L in this program, as well as an approximation ratio (i.e., approximate x iterations out of y). The algorithm first determines if the loop has the proper syntactical and semantical structure that will allow Approximate Unrolling to approximate it (line 1).

```
Data: $P$ an original program, $L$ a loop in $P$, $(x, y)$ determine the approximation
      ratio of $x$ iterations out of $y$
Result: $P'$ a version of $P$ with an approximate version of $L$
1 if is_syntax_semantic_fit(L) then
2     $\Gamma \leftarrow$ policy()
3     if $\Gamma \neq \emptyset$ then
4         unroll($L$,$y$)
5         approx_array($\Gamma, x$)
6         dead_code_elimination()
7     end
8 end
```

Algorithm 1: Approximate unrolling transformation process

If it does, then the algorithm proceeds in a sequence of three steps: (i) use a policy to determine which approximation strategy fits best (line 2, we present two strategies later in this section), (ii) unroll the loop y times (line 4), (iii) transform x assignments to an array assignment (line 5), (iv) remove code that has become dead code as a consequence of the approximation (line 6). In the following paragraphs we discuss each step. We illustrate the transformation process where we approximate 1 iteration out of 2 (i.e., $(x, y) = (1, 2)$).

Step 1. Loop Structure: The first step determines if a loop has the right syntax and semantics that allows Approximate Unrolling to optimize it. We consider `‘for’` loops that (i) have an induction variable incremented by a constant stride, (ii) contain an array assignment inside their body, and (iii) the indexing expression of the array assignment is value-dependent on the loop's induction variable.

Listing 3.1 gives an example of such loops. The example is a `‘for’` loop where (i) the induction variable `‘i’` is incremented by a constant stride, i.e. `‘i++’`. (ii) `‘A[i]’` is an array assignment inside the loop's body and (iii) the indexing expression of `‘A[i]’` is value-dependent on the induction variable `‘i’`.

Step 2. Approximation Strategy Policy: The policy determines the approximation strategy to use, if any. It performs the following analysis: say $|O|$ represents the number of instructions that can be removed from the original code, while $|LI|$ and $|NN|$ represent the number of instructions needed to perform the LI and NN interpolations respectively. If $|O| > |LI|$ (i.e. more instructions will be removed that inserted using the LI interpolation), resp. $|O| > |NN|$, the policy indicates that LI, resp. NN, should be used to approximate the loop. It is possible for a loop's body to be so small that both $|O| < |LI|$ and $|O| < |NN|$, meaning that the loop will in fact execute more instructions if approximated. In this case, the policy indicates that the optimization should refrain from optimizing the loop at all.

Notice that if both $|O| > |LI|$ and $|O| > |NN|$ are true, LI is preferred since it is more accurate in 70% of the cases in our dataset.

Step 3. Loop Unrolling: If the policy predicts that the loop will gain performance, the optimization unrolls the loop, just like a regular optimizing compiler would perform

```
for (int i = 0; i < N; i++) {
  phase = phase + phaseInc;
  double v = Math.sin(phase) *
      ampl;
  A[i] = v; }
```

Listing 3.1: Illustrative Example: A loop mapping a sine wave into an array.

```
for (int i = 0; i < N; i+= 2) {
  phase = phase + phaseInc;
  double v = Math.sin(phase) *
      ampl;
  A[i] = v;
  // UNROLLED ITERATION:
  phase = phase + phaseInc;
  double v1 = Math.sin(phase)*
      ampl;
  A[i + 1] = v1; }
```

Listing 3.2: Approximate Unrolling first unroll the loop as Loop Unrolling would normally do. In this listing, the Illustrative Example is unrolled using Loop Unrolling.

Loop Unrolling. The parameter y of the algorithm determines the number of times the loop is unrolled. For $y = 2$, this results in the loop of Listing 3.2 (an intermediate step we show for clarity).

Step 4(A). Approximating the loop with linear interpolation: If the policy determines that LI should be used, Approximate Unrolling adds code to the loop's body that interpolates the computations of the odd iterations as the mean of the even ones. Thus, the computations of iteration one are interpolated using the results of iterations zero and two, computations of iteration three are interpolated using the results of two and four and so on. Initially, Approximate Unrolling peels iteration zero of the loop. Then, it modifies the initialization and updates the statements to double the increment of the loop's counter variable. The approximate loop is terminated earlier and a guard loop is added to avoid out-of-bounds errors. Listing 3.3 shows the example loop using linear interpolation. Notice that some computations have been deleted. This is actually done in Step 5.

Step 4(B). Approximating the loop with nearest neighbor: If the policy determines that NN should be used, Approximate Unrolling modifies the update statement to double the increment of the loop's counter variable. It also adds code with the nearest neighbor interpolation at the end of the loop's body. The loop is also terminated earlier and a guard loop is also added to avoid out-of-bounds errors. Listing 3.4 shows the loop interpolated using nearest neighbor. Again, some computations are removed in the example, as discussed in the next step.

Step 5. Dead code removal: Once the approximate transformation is performed, some code is left dead (i.e. its computations are not longer used). In our example, the computation of the sine (`'v1=Math.sin(phase)*ampl'`) is no longer needed in the interpolated iteration once the interpolation code is added. This last transformation step removes this unused code. The expected improvement in performance comes precisely

```
//Initial iteration peeled:
phase = phase + phaseInc;
double v0 = Math.sin(phase) *
    ampl;
A[0] = v0;
int i = 1
if ( N > 2 )
  for (i = 2; i < N - 1; i += 2
      ) { // <-- Main Loop
    phase = phase + phaseInc;
    //v=Math.sin(phase)*ampl;
        <-- CODE REMOVED
    //Exact iteration:
    phase = phase + phaseInc;
    double v = Math.sin(phase) *
         ampl;
    A[i] = v;
    //Approximate iteration:
    A[i - 1] = (A[i] + A[i-2]) *
         0.5f; }
// Guard Loop:
for (j = i; j < N; j++) {
    phase = phase + phaseInc;
    double v = Math.sin(phase) *
         ampl;
    A[i] = v;}
```

Listing 3.3: After unrolling, our optimization approximates the unrolled iteration. The loop of Listing 3.2 is shown here, transformed by Approximate Unrolling using Linear Interpolation.

```
for ( i = 0; i < N - 1; i += 2 )
     {
  phase = phase + phaseInc;
  double v = Math.sin(phase) *
      ampl;
  A[i] = v;
  phase = phase + phaseInc;
  //v1=Math.sin(phase)*ampl; <--
       REMOVED
  A[i + 1] = A[i]; }
// Guard Loop:
for (j = i; j < N; j ++) {
  phase = phase + phaseInc;
  double v = Math.sin(phase) *
      ampl;
  A[i] = v;}
```

Listing 3.4: The loop of listing 3.2, transformed using nearest neighborg interpolation

from this removal, as less code is executed. After code removal, the example loop transformed using linear interpolation looks like the one in Listing 3.3, while the same loop approximated with nearest neighbor looks like the one in Listing 3.4. The output of these three loops is depicted in Figure 3.1.

Note on error accumulation: Is important to notice that only dead code resulting from the approximation is removed. Therefore, loop-carried dependencies are not removed, since their values are needed in the following iteration. This provides a guarantee that the exact iteration of the loop will always produce precise results. Therefore, error does not accumulate as the loop executes in the approximate iterations.

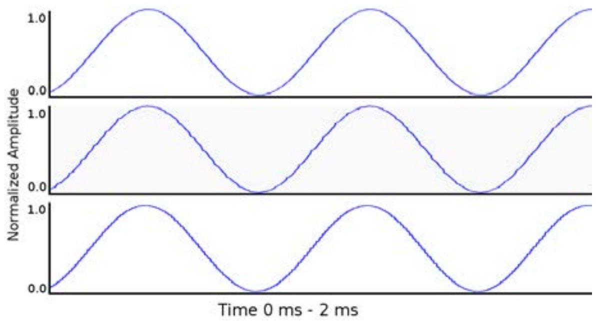


Figure 3.1: Sine waves generated by the motivation examples. The upper wave is generated by the loop of Listing 3.1, while the middle and lower ones are generated by loops of Listing 3.3 and 3.4, respectively.

3.1.2 Target Loop's Syntax

Approximate Unrolling targets the subset of `'for'` loops that (i) have an update expression that increments or decrements a variable by a constant value (ii) contain an array assignment inside its body and (iii) that have an indexing expression of the array assignment that is value-dependent on the loop's update expression.

We now use Backus-Naur form to formalize the syntax of the loops abiding to (i) and (ii), while (iii) is formalized using operational semantics in 3.1.3. For the sake of simplicity, we only show the case in which the variable is being incremented. It should not be difficult for the reader to derivate these rules for the case in which the variable is being decremented.

$\langle target\_for \rangle$ ::= '`for`' '`(`' $\langle expression \rangle$ '`;`' $\langle expression \rangle$ '`;`' $\langle update\_expr \rangle$ '`)`' $\langle t\_body \rangle$

$\langle update\_expr \rangle$ ::= $\langle identifier \rangle$ '`+=`' $\langle integer \rangle$ | $\langle identifier \rangle$ '`++`'

$\langle t\_body \rangle$ ::= '`{`' $\langle statement \rangle$ $\langle array\_assign \rangle$ $\langle statement \rangle$ '`}`'

$\langle array\_assign \rangle$ ::= $\langle expression \rangle$ '`[`' $\langle expression \rangle$ '`]`' '`=`' $\langle expression \rangle$

Grammar 3.1: Grammar of for loops targeted by Approximate Unrolling

Grammar 3.1 shows the BNF rules defining the loops Approximate Unrolling can target. In the grammar, the non-terminal rules $\langle expression \rangle$, $\langle identifier \rangle$, $\langle integer \rangle$ and $\langle statement \rangle$ define the Java expressions, identifiers, integer literals and statements respectively. Rule $\langle target\_for \rangle$ describes the loops targeted by Approximate Unrolling. The rule introduces two other non-terminals. The first non-terminal: $\langle update\_expr \rangle$ designates the update expression of the loop, consisting of a variable being incremented by a literal value, while $\langle t\_body \rangle$ designates a list of statements containing at least one array assignment, which is defined with the $\langle array\_assign \rangle$ non-terminal.

$$i, n \in \mathbb{Z} \qquad B \in \{\texttt{expression}\} \qquad [\![B]\!] \in \{true, false\} \quad t\_body = C$$

$$I_0, C \subset \{\texttt{statement}\} \quad \xi = \langle B, i := i + n, C \rangle$$

$$Ai = \{a \in \texttt{array\_assign} \mid \in REACH(\texttt{i := i + n})\} \quad \Gamma_N(C) \to \mathbb{R} \quad \Gamma_L(C) \to \mathbb{R}$$

Figure 3.2: Common definitions and rules used by both linear and nearest neighbor transformations

$$\text{For}_L\ \frac{(\Gamma_L(C) > 0) \Rightarrow true \quad (C \cap A_i \neq \emptyset) \Rightarrow true}{(For_L(I_0, B, i := i + n)\, C, \sigma) \to (I_0; \textbf{if } (B)\, Pre(\xi) \textbf{ else SKIP}, \sigma)}$$

$$\text{Ex}_L\ \frac{}{(Ex_L(\xi), \sigma) \to (i := i + 2 * n; \textbf{if } (B)\, C; Aprx_L(\xi) \textbf{ else } Post(\xi), C, \sigma)}$$

$$\text{Aprx}_L\ \frac{}{(Aprx_L(\xi), \sigma) \to (a[i - n] := (a[i] + a[i - 2 * n]) * 0.5; Ex_L(\xi), \sigma)}$$

$$\text{Pre}\ \frac{}{(Pre(\xi), \sigma) \to (C; Ext_L(\xi), \sigma)}$$

$$\text{Post}\ \frac{}{(Post(\xi), \sigma) \to (i := i - n; \textbf{if } (B)\, C; i := i + n;\ \textbf{else SKIP}, \sigma)}$$

Figure 3.3: Small step SOS of Approximate Unrolling using linear interpolation

3.1.3 Operational Semantics of Approximate Unrolling

Figures 3.3 and 3.4 show the operational semantics for Approximate Unrolling in small step operational style. The rules in Figure 3.3 specify the transformations for the linear interpolation, while the rules in Figure 3.4 specify the transformations for the nearest neighbor. Figure 3.2 provides a group of rules and definitions used by both transformations. As with the syntactic part, to keep the rules simple, we only describe the case in which the variable is incremented.

As defined in Figure 3.2, B is a boolean Java expression and I_0, C are sets of Java statements. I_0 represents the initialization statements, while C represents the loop's body. The tuple ξ is a syntactical sugar to make the rules more compact. Also, $REACH(X)$ is the reaching definitions set data-flow equation for a given expression

$$\text{For}N\,\frac{\Gamma_L(C) > 0) \Rightarrow false \quad (\Gamma_N(C) > 0) \Rightarrow true \quad (C \cap A_i \neq \emptyset) \Rightarrow true}{(For_N(I_0, B, i := i + n)\, C, \sigma) \rightarrow (I_0; Ex_N(\xi), \sigma)}$$

$$\text{Ex}_N\,\frac{}{(Ex_N(\xi), \sigma) \rightarrow (\textbf{if } (B)\, C; Aprx_N(\xi),\ \textbf{else SKIP}, \sigma)}$$

$$\text{Aprx}_N\,\frac{}{(Aprx_N(\xi), C, \sigma) \rightarrow (i := i + n; \textbf{if } (B) a[i] := a[i-1]; i := i + n; Ex_N(\xi)\ \textbf{else SKIP}, \sigma)}$$

Figure 3.4: Small step SOS of Approximate Unrolling using nearest neighbor

[CT05]. Functions $\Gamma_L(C)$ and $\Gamma_N(C)$ are implementation-specific functions that tell us if there is going to be any gain from transforming the loop using the linear or nearest neighbor interpolations respectively. These functions are the representation of the policy in the formal rules. We define these functions for our implementation in section 3.2. Finally, a is an array assignment statement conforming to the $\langle array\_assign \rangle$ syntactical rule and A_i is the set of array assignment expressions contained in the reaching definitions set of the update expression of the loop (i.e. $REACH(X)$). We use the common definitions for rules SKIP, ASSIGNMENT, SEQUENCE and IF.

3.1.3.1 Rules For the Linear Transformation

In this subsection we describe the semantic rules of the linear transformation as follows:

For$_\text{L}$. The For_L rule interprets a loop using linear interpolation. The rule reduces to the interpretation of the loop's initialization statement I_0. If predicate B holds, rule Pre is interpreted. The rule reduces only if the gain function is positive ($\Gamma_L(C) > 0$) and the body C of the loop contains an array assignment being indexed by the loop's induction variable (i.e. $C \cap A_i \neq \emptyset$).

Pre. As shown in Listing 3.2 the loops approximated using linear interpolation have iteration zero peeled. This rule represents such initial peeled iteration. It reduces to the interpretation of the loop's body once, without any modifications. Then, the exact part of the loop (described by rule Ext_L) is called.

Ext$_\text{L}$. This rule represents one loop's exact iteration. The odd iteration is skipped by doubling the incrementing of the counter variable ($i := i + 2n$), then if predicate B still holds, an exact iteration of the loop is executed. Next, the code approximating intermediate values is called in rule $Aprx_L$.

Aprx$_\text{L}$. Approximates the value of odd arrays slots. The approximation is found as the mean of the previous and following values of the approximate slot, the value of the even iteration array slot is calculated as:

$a[i - n] := (a[i] + a[i - 2n]) * 0.5.$

Notice that the counter's variable i was incremented twice to skip the approximate iteration in the exact one, therefore the odd slot must be accessed now using index $i-n$ and the previous even slot using index $i-2n$.

Post As the initial iteration is peeled and the counter's variable value is doubled to unroll the loop, if the number of iterations in the original loop is even, the last iteration of the loop is not executed. To solve this, one last iteration is peeled and appended after the loop.

3.1.3.2 Rules For the Nearest Neighbor

Similarly to what we did for the linear transformation, in this section we describe the semantic rules for the nearest neighbor transformation.

For$_\mathrm{L}$. The For_L rule transforms a loop using nearest neighbor interpolation. The rule can be reduced to the execution of the initialization statements I_0, followed by the execution of Ex_N. The rule reduces only if the gain function is for the linear interpolation is negative ($\Gamma_L(C) \leq 0$), the gain function for the nearest neighbor is positive ($\Gamma_L(C) > 0$) and the body C of the loop contains an array assignment being indexed by the loop's induction variable (i.e. $C \cap A_i \neq \emptyset$).

Ex$_\mathrm{N}$ This rule executes an exact iteration of the loop if the predicate B holds, then it calls $Aprx_N$.

Aprx$_\mathrm{N}$ The rule increments the counter variable, approximates one array slot by setting its value equal to the one calculated before and then it increments the variable again. Finally Ex_N is called. Unlike linear interpolation, the rule is supposed to approximate the value of the array assigned in the even iteration.

3.2 Implementation

This section describes our experimental implementation of Approximate Unrolling in the C2 compiler of Hostpot. The Hostpot V.M. is used by billions of devices today. It come packed with two compilers: C1 (or Client) and C2 (or Server). We chose to implement Approximate Unrolling directly in the C2 compiler to avoid phase ordering problems, to reuse the C2's value dependency graph that we needed for our analysis and to obtain guarantees that Approximate Unrolling can actually improve the performance of a highly optimized code created by production-ready compiler. Also, by implementing the optimization in C2, we provide support for other JVM languages such as Scala.

3.2.1 Approach Overview

Approximate Unrolling is a machine-independent optimization. This kind of optimization operates by reshaping the Ideal Graph, which is the internal representation (IR) of the C2 compiler.

In short, our implementation works as follows: the Ideal Graph contains nodes representing operations very close to assembler instructions. Nodes are linked by edges representing data dependencies. There is a node type called `Store` that represents a storage to memory. Our transformation is performed after the compiler performs the 'regular' Loop Unrolling. During the unrolling, each node in the loop's body gets duplicated, resulting in one extra node for each unrolled iteration. Let us consider `StoreA`, a node representing a value storage into memory owned by an array. Before the unrolling, there is only one node `StoreA`. After the unrolling, there are two nodes: `StoreA` and `StoreB`. If we find nodes like `StoreA` and `StoreB` in a loop body, we say that the loop has the correct structure and we pass it to the policy function.

If the policy recommends Nearest Neighbor, we disconnect one of the `Store` nodes (say `StoreA`) of all its value-dependencies and connect to the value-dependencies of the other (say `StoreB`), resulting in both `Store` storing the same value in two different array slots.

If the policy recommends Linear Interpolation the process is very similar to Nearest Neighbor, the only difference is that some nodes representing the mean computation are added to the input of `StoreA` to perform the mean calculation.

Once we finish transforming the graph, we delete of all nodes on which `StoreA` was originally value-dependent, removing most computations of one of the two unrolled iterations.

In the rest of the section we expand on this process. First, we describe the Ideal Graph, the internal representation of the C2. Then, we go into the details of our implementation and exemplify it using a toy loop. Our modified version of the Hostpot JVM is available on the webpage of the Approximate Unrolling project [1]

3.2.2 The Ideal Graph

The Ideal Graph (IG) [CP95] is a Program Dependency Graph [FOW87]. All C2's machine independent optimizations work by reshaping this graph. In the Ideal Graph, nodes are objects and hold metadata used by the compiler to perform optimizations. Nodes in this graph represent instructions that are as close as possible to assembler language (i.e. `AddI` for integer addition and `MulF` for float multiplication).

The IG metamodel has nearly 380 types of nodes. In this section, we deal with five of them to explain our technique: `Store`, `CountedLoop`, `Phi`, `Add` and `Mul`. `Store` nodes represent storages into memory. They contain metadata indicating the type of variable holding the memory being written, making it easy to identify `Store` nodes writing to arrays. The Ideal Graph is in Static Single Assignment (SSA) form and the `Phi` nodes represent the ϕ functions of the SSA. `CountedLoop` represents the first node of all the loops that Approximate Unrolling can target. The `CountedLoop` type contains two important metadata for our implementation: a list of all nodes representing the loop's body instructions and a list with all nodes of the update expression. Finally, `Add` and `Mul` nodes represents the addition and multiplication operations.

Nodes in the IG are connected by control edges and data edges. Yet, data edges are the most important ones for our implementation, and we will refer to this kind of edges exclusively, unless noted otherwise. Data edges describe value dependencies, therefore

[1] `https://github.com/approxunrollteam`

the IG is also a directed Value Dependency Graph [AKPW83]. If a node B receives a data edge from a node A, it depends on the value produced by A. Edges are pointers to other nodes and contain no information. Edges are stored in nodes as a list of pointers. The edge's position in the list usually matters. For example, in `Div` (division) nodes the edge in slot 1 points to the dividend and the edge in slot 2 points to the divisor.

The `Store` requires a memory address and a value to be stored in the address. The memory edge e_M is stored at slot 2 and the value edge e_V at slot 3 of the edge's list. Edge e_M links the `Store` with the node computing the memory address where the value is being written, while e_V links the `Store` with the node producing the value to write.

Let us consider the very simple example of listing 3.5,. The resulting IG for this loop is shown in figure 3.5. In the figure, the `StoreI` represents the assignment to `A[i]`, the `MulI` node represents the `i*i` expression. The address is resolved by the nodes in the Cluster A (containing the `LShift` node).

```
for ( int i = 0; i < N; i++ )
  A[i] = i * i;
```

Listing 3.5: Example loop for the implementation

3.2.3 Detecting Target Loops

Sections 3.1.2 and 3.1.3 formally described the shape of loops targeted by the proposed optimization. This section describes how our implementation detects them in the IG.

Section 3.1.2 defined the loops Approximate Unrolling can target. Fortunately, Java `for` loops having a constant-increment update expression are also the target of other well known optimizations such as Range Check Elimination and Loop Unrolling. Therefore, the C2 recognizes them and marks their start using `CountedLoop` nodes. When our optimization starts, the compiler has already marked the loops with `CountedLoop` nodes, recognized the nodes belonging to the update expression and the ones belonging to the loop's body. The compiler does this using the work described in [Cli95, Vic94, Tar]. Figure 3.5 shows the `CountedLoop` recognized by the C2 for the loop in listing 3.5. The nodes in the graph are those listed in the `CountedLoop` metadata. This metadata also indicates that the update expression contains solely the `AddI` node (in gray).

Once the loops with constant-increment update expression are detected, the next step consists in determining if there is an array whose index expression value depends on the loop's update expression. As we mentioned, the `CountedLoop` node maintains a list of all the nodes in its body. To determine if there is an array assignment within the loop, we search this list looking for a `Store` writing to memory occupied by an array. In the example of figure 3.5 we find the `StoreI` node (in gray).

Finally, we check if the array index expression value depends on the loop's update expression. As the IG is a Value Dependency Graph, we look for a path of data edges between the `Store` node representing the array assignment and any node belonging to the loop's update expression. In the example of Figure 3.5 this path is highlighted using bold gray edges. Thanks to the metadata stored in `CountedLoop`, we know that the

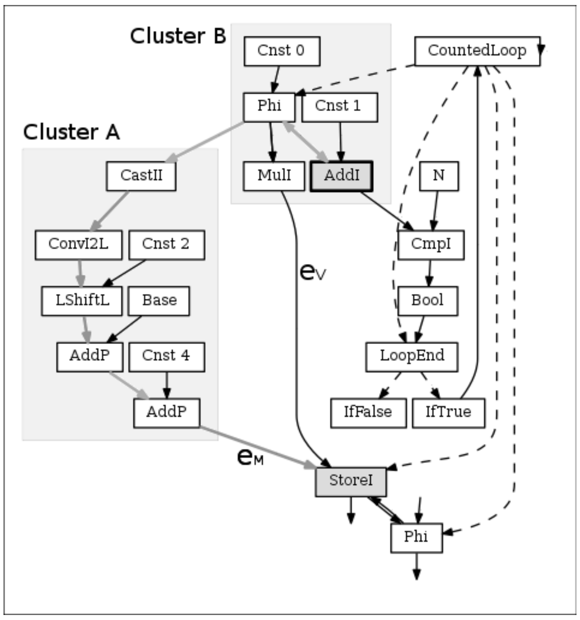


Figure 3.5: The ideal graph of the example loop of listing 3.5. Dashed arrows represent control edges and solid arrows represents data edges.

update expression only contains the `AddI` node. Therefore, in the example, the path is composed of the following nodes: `AddI` → `Phi` → `CastII` → `ConvI2L` → `LShiftL` → `AddP` → `AddP` → `StoreI`.

3.2.4 Unrolling

Our implementation piggybacks in two optimizations already present in the C2 compiler: Loop Unrolling and Range Check Removal. At the point Approximate Unrolling begins, the compiler has already unrolled the loop and performed Range Check Removal.

While unrolling, the compiler clones all the instructions of the loop's body. Figure 3.6 shows the IG once the loop of listing 3.5 has been unrolled. The cloning process introduces two `Store` nodes: `StoreI` and `StoreI-Cloned`. Due to C2's design, the cloned nodes belong to the even iteration of the loop. Once the loop is unrolled, Approximate Unrolling reshapes the graph to achieve the interpolated step by modifying one of the two resulting iterations. Nearest Neighbor modifies the even iteration, while Linear interpolation reshapes the odd iteration.

3.2.4.1 Nearest Neighbor Interpolation

As mentioned in section 3.2.2, a `Store` node takes two input data edges e_M and e_V. Edge e_M links with the node computing the memory address, while e_V links with node producing the value to write.

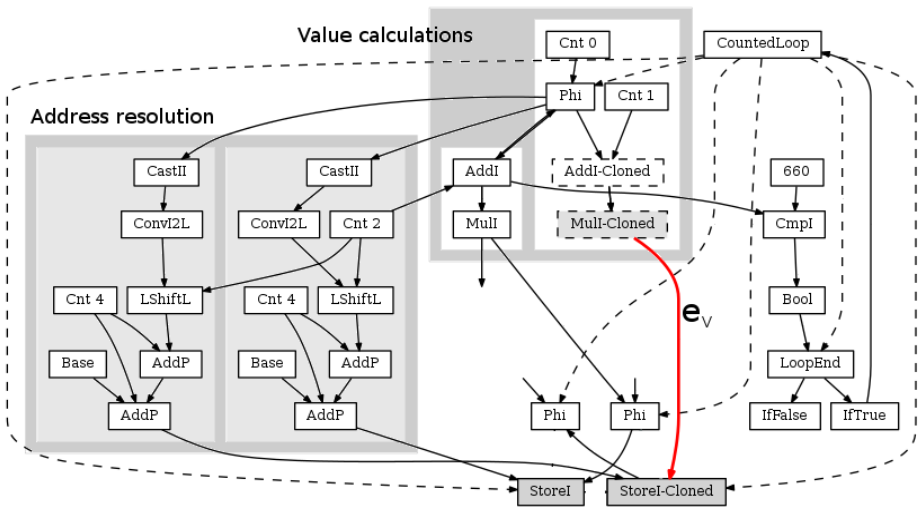


Figure 3.6: The ideal graph for the unrolled loop of listing 3.5 before approximating. Solid arrows represent data edges.Notice nodes `MulI-Cloned` and `AddI-Cloned`, which produces the even iteration's value. Edge e_V connects these nodes with `StoreI-Cloned`.

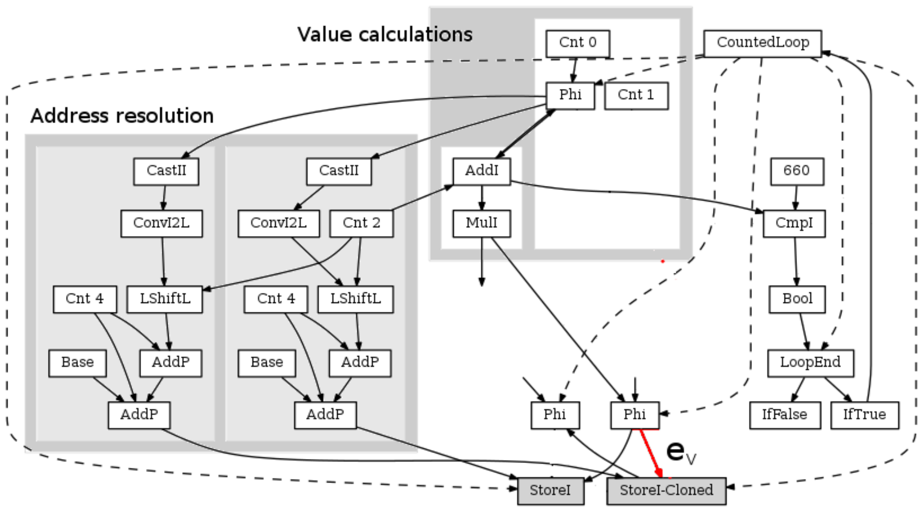


Figure 3.7: The ideal graph for the unrolled loop of listing 3.5 after Approximate Unrolling has modified the graph using nearest neighbor interpolation. Notice the removal of the `MulI-Cloned` and `AddI-Cloned` nodes. These nodes will not translate into assembler instructions anymore. Also, data edge e_V now connects `StoreI-Cloned` with the same `Phi` node that produces a value for `StoreI`, meaning that the same value is going to be stored in two different locations.

Approximate Unrolling performs nearest neighborhood interpolation by disconnecting the cloned `Store` node from the node producing the value being written (i.e. it deletes e_V). In figure 3.6 this means to disconnect node `MulI-Clone` (in gray) from node `StoreI-Clone` by removing edge e_V.

This operation causes the node producing the value (in the example `MulI-Clone`) to have one less value dependency and potentially become dead if it has no other dependencies. A node without value dependencies means that its computations are not being consumed and therefore becomes dead code. In this case, the node is removed from the graph. We recursively delete all nodes that do not have dependencies anymore, until no new dead nodes appear. In figure 3.6, we delete `MulI-Cloned` and then `AddI-Cloned`. This simplification of the IG translates into fewer instructions when the IG is transformed in assembler code.

After the removal, Approximate Unrolling connects the node producing the value for the original `Store` into the cloned `Store`. Figure 3.7 shows the shape of the IG after Approximate Unrolling has approximated the graph using nearest neighbor. Note that `MulI-Clone` and `AddI-Clone` are deleted and that `Store-Clone` is connected by e_v to the same node as `StoreI`. The nodes producing the address remain different.

Listing 3.6 shows the code generated by C2, without performing Approximate Unrolling: the compiler has unrolled the loop twice, generating four storages to memory (lines 2, 3, 9, 10) and four multiplication instructions (`imull`, lines 5, 8, 12, 17). Listing 3.7 shows the code generated for the same loop using our transformation: there are still four storages (lines 4, 5, 9, 10), but only two multiplications (Lines 8, 13).

```
B8: #
movl  [RCX + #16 + R9 << #2], R8
movl  R9, R10 # spill
addl  R9, #3  # int
imull R9, R9  # int
movl  R8, R10 # spill
incl  R8      # int
imull R8, R8  # int
movl  [RCX + #20 + R10 << #2],
   R8
movl  RDI, R10        # spill
addl  RDI, #2 # int
imull RDI, RDI        # int
movl  [RCX+#24+R10<< #2], RDI
movl  [RCX+#28+R10<< #2], R9
addl  R10, #4 # int
movl  R8, R10 # spill
imull R8, R10 # int
cmpl  R10, R11
jl,s  7
```

Listing 3.6: Assembler code generated for the example loop without using Approximate Unrolling

```
B7: #     B8 <- B8  top-of-loop
   Freq: 986889
movl  RBX, R8 # spill
B8: #
movl  [R11+#16 + RBX<<#2], RCX
movl  [R11+#20 + R8<<#2], RCX
movl  RBX, R8 # spill
addl  RBX, #2 # int
imull RBX, RBX # int
movl  [R11+#24+R8<<#2], RBX
movl  [R11+#28+R8<<#2], RBX
addl  R8, #4  # int
movl  RCX, R8 # spill
imull RCX, R8 # int
cmpl  R8, R9
jl,s  B7
```

Listing 3.7: Assembler code for the example loop using Approximate Unrolling. Notice the consecutive `movl` instructions storing the same value to consecutive addresses

3.2.4.2 Linear Interpolation

To unroll using linear interpolation, Approximate Unrolling needs the first and last iteration of the loop peeled. Fortunately, this is also a requirement of other optimizations such Range Check Removal and we exploit this feature to peel the first and last iterations of the loop. The current implementation of the Range Check Removal creates two guard loops, one before the main loop and other after. These guard loops ensure that the main loop will not go out of bounds on the array being assigned. We exploit these two guard loops for the linear interpolation.

Approximate Unrolling performs the LI interpolation following a process similar to nearest interpolation. The differences are that it disconnects the value data edge e_v from the original `Store`, rather than the cloned `Store`. This is because C2's design implies that the cloned nodes belong to the even iteration, but linear interpolation approximates odd iterations. After the value data edge is disconnected, some nodes become dead. Here we use the same process to remove unused nodes. Finally, the interpolation is performed in the following way: (i) a `Add` node is created that receives as input the output of the cloned `Store` and a `Phi` node representing merge between the previous odd iteration of the loop and the current one (ii) a `Mul` node is created to multiply the result of this addition by 0.5 and this node is connected to the the original `Store` effectively interpolating the loop's even iteration.

```
   Data: $store$ is the IG node representing the storage to an approximate array slot
   Result: A value indicating whether to approximate this loop or not.
1  $|O| \leftarrow 0$
2  $rem$.pushAll($store$.incoming_nodes())
3  $visited$.addAll($store$.incoming_nodes())
4  while $\neg$ $rem.empty()$ do
5      $n \leftarrow rem$.pop()
6      $out \leftarrow n$.outgoingEdgeCount() $-1$
7      if $\neg$ ($n \in visited \vee out > 0$) then
8          $rem$.pushAll(n.incomingNodes())
9          $visited$.add(n)
10         $|O| \leftarrow |O| +$ weight(n)
11     end
12 end
13 if $|LI| < |O|$ then
14     return use_LI
15 end
16 if $|NN| < |O|$ then
17     return use_NN
18 end
19
20 return use_NONE
```

Algorithm 2: Selection mechanism

3.2.5 Strategy Policy implementation

Approximate Unrolling uses a policy to determine which approximation strategy use to each loop. The policy uses three cost variables: $|O|$ represents the cost of executing the loops original's instructions, while $|LI|$ and $|NN|$ represents the cost of running the LI and NN approximation respectively.

Algorithm 2 shows how our policy's implementation works in the C2 compiler. Each node in the IG has a fixed cost weight defined manually by us. The amount and types of nodes inserted using LI and NN are fixed: three nodes (`Mul`, `Add`, `Store`) for LI and one node (`Store`) for NN. Therefore, we know beforehand the $|LI|$ and $|NN|$ interpolation's costs. We are only missing the cost $|O|$ of the original code. To compute this we begin with $|O| = 0$. We then perform a depth-first-search (DFS) in the IG starting from the `Store` node representing the array slot receiving the approximate value. For each node encountered during the DFS we compute a value *out* equal to the number of outgoing data edges minus one. Outgoing data edges represents value-dependencies in the IG. Since the original value will not be used in the approximate loop, the node will have one less value-dependency if the loop is approximated. Therefore, if $out - 1 = 0$ it means that the node will have no data-dependencies (i.e. will become dead code) and can be deleted in the approximated loop. If this is the case, we add the node's weight w to $|O|$ and we continue the DFS from this node traveling into its incoming edges. This is done because a node that becomes dead code can also cause other nodes to turn dead. On the contrary, if $out > 0$ then the node cannot be deleted, we do not add its weight to $|O|$ and we do not pursue the DFS from this node.

When the DFS is over, we have computed the value of $|O|$. Then, the policy will indicate that LI should be used if $|LI| < |O|$, that NN should be used if $|LI| < |O| \wedge |NN| > |O|$, or that Approximate Unrolling should not optimize the loop if $|LI| > |O| \wedge |NN| > |O|$.

If the policy recommends not to optimize the loop, the optimization stops and no further transformation of the loop is performed.

3.3 Evaluation

The objective of our evaluation is to assess whether Approximate Unrolling can provide a satisfactory trade-off that maximizes performance gain and energy efficiency while minimizing accuracy losses. Besides, we want to know if the selection mechanism can determine the approximation strategy providing optimal results for each loop. We also expect to gain insights on the types of patterns on which Approximate Unrolling works best, and the impact of perforation ratio in performance and accuracy. Finally, we compare our optimization with Loop Perforation [SDMHR11], which is a state of the art loop approximation technique that works by completely skipping iterations of the loop. We investigate these aspects through the following research questions:

RQ1: Can Approximate Unrolling provide a satisfactory accuracy/energy-performance trade-off?

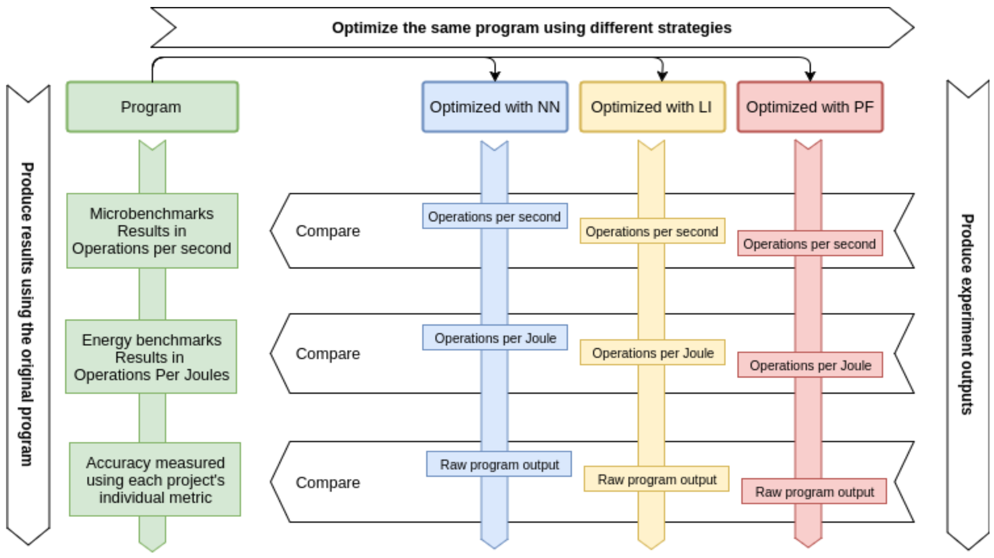

Optimize the same program using different strategies
Program
Optimized with NN
Optimized with LI
Optimized with PF
Produce results using the original program
Produce experiment outputs
Microbenchmarks Results in Operations per second
Energy benchmarks Results in Operations Per Joules
Accuracy measured using each project's individual metric
Compare
Operations per second
Operations per Joule
Raw program output

Figure 3.8: The figure shows our evaluation's overall process. Each program was optimized using three strategies. Then, the results obtained with each strategy were compared with those obtained using the program without approximation

If the accuracy losses are unacceptable, there is no point in making calculations faster. On the other hand, without significant increments in speed and energy efficiency, reducing accuracy may not pay-off. To answer this question, we use a set of microbenchmarks to measure speedups for each of the loops in our dataset. We then observe the energy consumption of these microbenchmarks to evaluate energy reductions. Finally, we evaluate accuracy losses using one specific accuracy metric for each project in our dataset. The metric used for each project is described in section 3.3.1.

RQ2: How does Approximate Unrolling compare with loop perforation?

Loop Perforation [SDMHR11] is a state-of-the-art approximate loop optimization. Hence, we compare our work with it. Our intuition is that these two optimizations are complementary and work best in different situations. To learn more about this, we compare both in terms of speed gain, energy efficiency, accuracy loss and trade-off.

RQ3: How does the ratio of approximation influence the trade-offs?

Approximate Unrolling can use different ratios of approximation. In the examples, formalization and the explanation of the implementation, we show approximation in 1 out 2 iterations. However other ratios are also valid and we must understand the approximation ratio's impact on the approximation's quality.

RQ4: What are the situations in which Approximate Unrolling works best?

To further understand what are the loops that can benefit from Approximate Unrolling and how our approach could be applied beyond the four case studies analyzed here, we perform a qualitative classification of the loops that we approximate.

Table 3.1: Case studies for our experiments

Case Study	Domain	Workload	Accuracy Metric	Loops	**Approximable**
OpenImaJ	Multimedia analysis	Face Recognition System	Dice	118	16
Jsyn	Musical Synthesizers	3xOsc synthesizer clone.	PEAQ [ŠBD05]	8	8
Lucene	Text search engine	Text Search Application	Baek [BC10]	151	9
Smile	Machine Learning	ZipCode digit classifier	Recall	73	12

3.3.1 Case Studies

To evaluate our approach we use four libraries belonging to different domains: OpenImaJ [HSD11], Jsyn[Bur98], Apache Lucene[MHG10] and Smile[Hai17].

We chose libraries from multiple domains (video and sound processing, search engines and machine learning) in which performance played an important role. Other parameters we considered when selecting our case studies was the codebase's size, the test suite's quality and the project's popularity. Therefore, all projects in our dataset are large, well tested and popular (as measured in Github stars).

We evaluate OpenImaJ using a workload that performs real-time face recognition in videos featuring people of different sex, age and ethnicity speaking directly to the camera. Jsyn is a framework to build software-based musical synthesizers. The workload to evaluate Jsyn builds a clone of the popular 3xOsc[2] synthesizer and use it to render a set of scores into WAV sound files. Lucene [3] is a text search engine written entirely in Java. Lucene's workload indexes a set of text files and then returns the results of multiple text search performed over these files. Smile [4] is a machine learning library. The workload to assess Smile consisted in a classifier that was able to learn and then recognize a dataset of handwritten zip-codes digits.

Table 3.1 shows for all case studies in our dataset, their domain and workload. The table also shows the number of `'for'` loops covered in each library by the workload ('Loops') and the subset of these loops Approximate Unrolling can target (Approximable). In the table, column 'Accuracy Metric' shows the metric we used to measure accuracy losses, we expand on these metrics in 3.3.3.

[2] `https://www.image-line.com/support/FLHelp/html/plugins/3x%20OSC.htm`
[3] `https://github.com/apache/lucene-solr`
[4] `http://haifengl.github.io/smile/`

3.3.2 Methodology

Column 'Approximable' of Table 3.1 shows the number of loops that Approximate Unrolling could target in a case study that were also covered by the workload's execution. For each of these loops we measured performance, energy consumption and impact on application's accuracy. We repeated the same observations for the approximated versions of all loops. This was done using both approximation strategies (LI and NN).

Performance was measured using microbenchmarks. This technique allows observing the loop's performance improvement caused by our optimization in isolation, without the added noise of the rest of the program. Also, microbenchmarks are widely used by the authors of the C2 to measure their own optimizations [Ale13b], as shown by the Oracle's bug database[5]. Determining whether there is an effective gain in terms of performance is a challenging task in Java [Ale13b]. We run our microbenchmarks using a statistical methodology [GBE07] to ensure that the measurements of each run were consistent. All runs were performed in an Intel i7 i7-6600U CPU, 2.60GHz with 16GB RAM running Linux Ubuntu 16.04.

Energy was assessed by estimating the total CPU energy consumption of our microbenchmarks using JRALP [LPYD15], a Java library that exposes to Java programs the Intel's Running Average Power Limit (RAPL) Technology [6]

Finally, we observed each loop's impact on its respective application's accuracy. To do so, we first ran the workload without any approximation to obtain an output that we use as reference. In a second step, we ran several times the same workload approximating only one of the N approximable loops covered by the workload each time. For all loop we repeated this process three times, each time using a different approximation strategy: Nearest Neighbor, Linear interpolation and Loop Perforation. This yielded $N \times 3$ different outputs for a given workload that we then compare against the reference output using a metric specific to the application's domain.

These experiments were made overriding the default's Approximate Unrolling strategy selection policy. By overriding the policy we forced the optimization to use the LI, NN and Loop Perforation strategies one after the other in all cases to observe their respective impact in all the loops of our dataset.

3.3.3 Metrics

3.3.3.1 Trade-off metric

In order to objectively assess the speedup-energy/accuracy trade-off we must quantify it. To do this we define a trade-off metric in Equation 3.1. Variable P is the performance improvement of the loop, measured as the ratio of operations per second of the approximated loop's microbenchmark over the operations per second of the precise loop's microbenchmark; $||L||$ is the normalized value in the range [0..1] of the accuracy metric's value L for each application. Finally, E is the ratio of energy consumption of the precise loop's microbenchmark over the approximate loop's microbenchmark.

[5]`http://bugs.java.com`. Bugs ids: 8152910, 8050142, 8151481 and 8146071

[6]`https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl`

$ACCEPTABLE(x)$ is a function that returns true if the losses are acceptable and false otherwise.

$$T(P, L, E) = \begin{cases} \frac{P+E}{2} \times (1 - ||L||), & \text{if } ACCEPTABLE(L) \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

The metric rewards performance improvements and penalizes accuracy losses and energy consumption. It gives no reward to a loop if accuracy losses are not acceptable. We assume that some accuracy reduction can be tolerated, but only to some degree. That is why the metric is divided in two cases, one for when the accuracy losses are acceptable and another for when it is not. If the losses are acceptable, we ideally want our approximate loops to increase performance and energy efficiency the most, while giving away as little accuracy as possible. Performance and energy values are weighted to compensate for the fact that these two parameters are highly related to the same variable, which is the amount of instructions executed. This metric allows us to reason about how good is the trade-off. For a non-approximate (precise) loop $T(1, 0, 1) = 1$. Therefore we consider a trade-off good only if $T > 1$ for a particular loop, since this means that we have increased performance to a higher degree than we have decreased accuracy.

3.3.3.2 Speed-up

In order to quantify the speedup, we use microbenchmarks to measure the number of operations per second (ops) of a particular piece of code. The speedup metric is defined as the ratio of operations per second of the optimized code over the original code as measured by a microbenchmark:

$$P(original, optimized) = \frac{ops(optimized)}{ops(original)}$$

3.3.3.3 Energy consumption efficiency gain

We also use use microbenchmarks to measure the CPU energy consumption in operations per Joule (opJ). We then define the energy efficiency gain as the ratio of operations per Joule of the original code over the optimized code:

$$E(original, optimized) = \frac{opJ(original)}{opJ(optimized)}$$

3.3.3.4 Accuracy metrics

Like other studies in Approximate Computing [SDF$^+$11a, BC10], we use a different metric L for each case study to improve the relevance of our observations. The reason for this is that in each domain the result's quality is perceived differently. Also, the degree of tolerance w.r.t. accuracy loss varies from case to case. Therefore, is not possible to choose a one-size-fits-all metric and we decided to provide one specific way of measuring accuracy in each application.

Jsyn Accuracy Metric: To pairwise compare the audio files produced by the optimized versions of the synthesizer with those produced using the unoptimized version, we use the Perceptual Evaluation of Audio Quality (PEAQ) [ŠBD05] metric. This is a standard metric designed to objectively measure the audio quality as perceived by the human ear. PEAQ takes one reference and one test file and compares them. It assigns a scale of continuous values to the to the test audio, depending on how well it matches the reference: 0 (non audible degradation), -1 (audible but not annoying), -2 (audible slightly annoying), -3 (annoying) -4 (completely degraded). The acceptable function is defined then as

$$ACCEPTABLE(x) \rightarrow x < -2.5$$

.

OpenImaJ. Accuracy Metric: We use the Dice metric: $D(A, B) = \frac{2*|A \cap B|}{|A|+|B|}$ to compare the pixel's set inside the rectangle detected by the application without approximation against the pixel' set obtained with the approximated version of the face recognition application. Dice is a set comparison metric with continuous values between 1 and 0, where 1 means identical sets and 0 means completely different sets. For most segmentation algorithms that use Dice, 0.85 is an acceptable value, therefore we define the acceptable function for this application as

$$ACCEPTABLE(x) \rightarrow x > 0.85$$

Lucene Accuracy metric: We use a similar metric to the one used in [BC10] to evaluate the impact of approximate loops in the Bing[7] search engine. We give 1 point to each hit that the approximate engine returned in the same order as the non-approximated one, 0.5 points if the hit is returned but in a different order and 0.0001 points if the hit is not in the original version. The metric's value is the average of all hits or 0.0 points if the program crashes. We define the acceptable function as

$$ACCEPTABLE(x) \rightarrow x > 0.95$$

Smile Accuracy metric: We use the classification's recall (i.e. the number of properly classified elements over the total number of elements). According to [Has] a 2% error is a good value for this dataset, so we define acceptable as

$$ACCEPTABLE(x) \rightarrow x > 0.98$$

3.4 Results

Figures 3.9 to 3.20 show the performance, energy and accuracy results of our experiments. In all figures, each loop is represented using three bars that show (from left to right) the results obtained using Nearest Neighbor (NN) with ratio 1/2, Linear Interpolation (LI) with ratio 1/2 and Perforation (PERF). Figures 3.9, 3.10, 3.11, 3.12 show the

[7] `http://www.bing.com/`

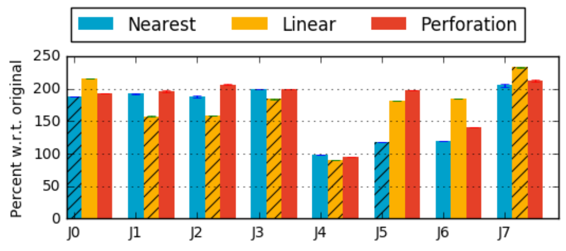


Figure 3.9: JSyn performance improvements

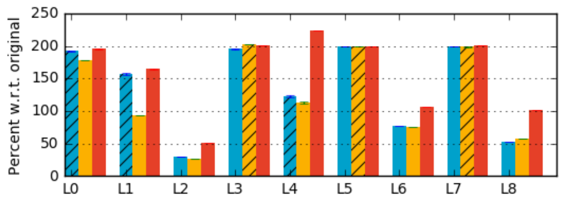


Figure 3.10: Lucene. Performance improvement

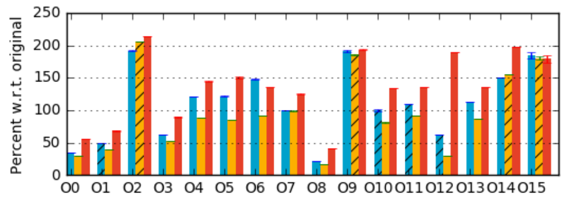


Figure 3.11: OpenImaJ. Performance improvement

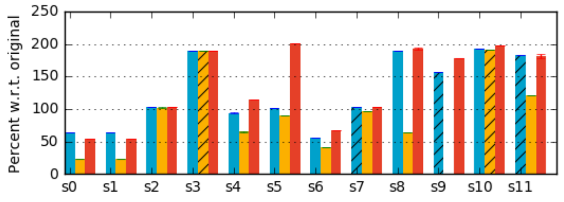


Figure 3.12: Smile. Performance Improvements

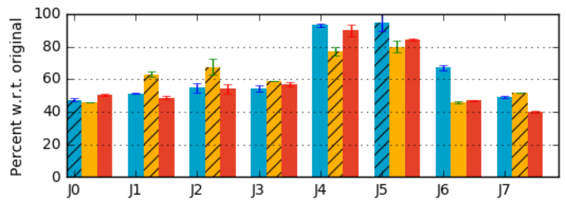


Figure 3.13: JSyn energy reductions

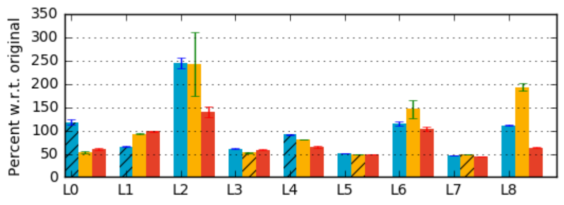


Figure 3.14: Lucene. Energy consumption

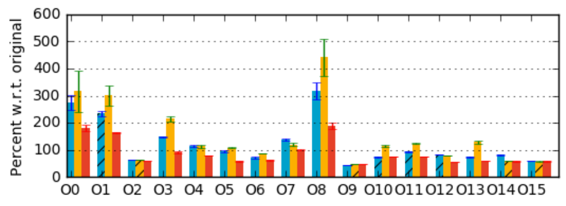


Figure 3.15: OpenImaJ. Energy consumption

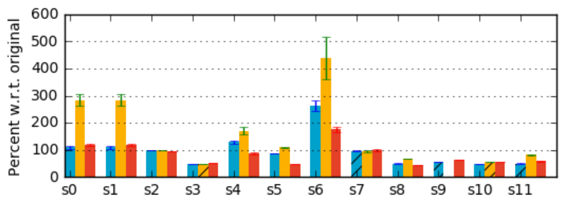


Figure 3.16: Smile. Energy consumption

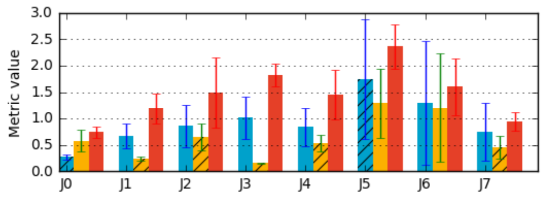


Figure 3.17: JSyn Accuracy metric

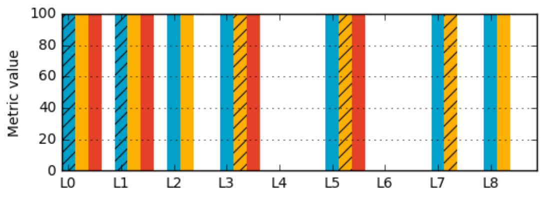


Figure 3.18: Lucene. Accuracy metric's value (Higher is better, zero means crash)

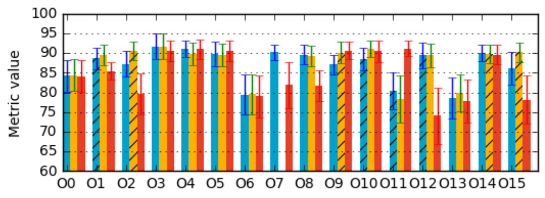


Figure 3.19: OpenImaJ. Accuracy metric's value

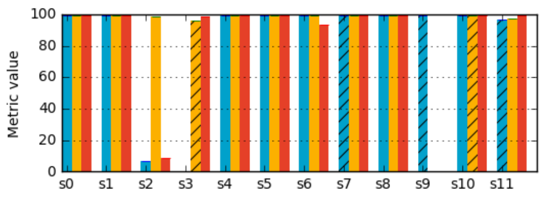


Figure 3.20: Smile. Accuracy metric's value

performance gains for each loop in our dataset. Figures 3.13, 3.14, 3.15, 3.16 represent the percent of energy consumption w.r.t to the precise version of the loop. Figures 3.17, 3.18, 3.19, 3.20 show the impact of each loop in its host application's accuracy. In the performance graphs, the Y axis represents the percent of gain w.r.t to the original loop. For example, a 200% value means that the approximate version of the loop run twice as fast. The same applies to energy consumption, a 50% value means that an approximate loop needs only half of the energy to finish than its precise counterpart. In the accuracy graphs, the Y axis represents the accuracy metric's value.

In Section 3.1.1, we illustrate how the policy recommends the approximation strategy to approximate a loop. The policy can also suggest not to optimize the loop at all. Stripped bars represents the strategy recommended to approximate a particular loop. For example, Figure 3.9 shows that the strategy proposed for loop J0 is NN while for loop J1 is LI. The policy also indicated not to approximate some loops in our dataset, for example, Figure 3.10 shows that this is the case for loops L2 and L8 (no striped bars). We include all loops in our dataset in the plots, even those rejected by policy. We do so to learn about the policy's quality and to observe the optimization's impact in all cases.

Tables 3.2 and 3.3 summarize the data presented in the figures. Table 3.2 shows the numbers of loops that improve performance and energy efficiency, while table 3.3 shows energy reductions and impact on accuracy of the loops in our dataset. In table 3.2, column 'Loops' show the number of loops amenable for Approximate Unrolling in our dataset, while columns 'Speedup' and 'Less Consumption'show the number of loops that actually improved speed and consume less energy respectively. Column 'Rec.' shows the number of loops the policy recommended to optimize and 'Rec. Speedup' how

Table 3.2: Performance and Energy Improvements for the loops of our dataset

Project	Loops	Speedup	%	Less Consumption	%	Rec.	Rec. Speedup	%	Rec. Less Consumption	%
JSyn	8	7	87.5	8	100	7	7	100	7	100
OpenImaJ	16	9	56.2	8	50	8	7	75	7	75
Lucene	9	6	66.6	6	66.6	6	6	100	6	100
Smile	12	5	41.7	6	50	5	4	100	4	100
Total	45	27	60	28	60	26	24	92	24	92

Table 3.3: Impact in accuracy of the loops in our dataset

Project	Loops	Accept	%	**Crash**	%	**Rec. Loops**	**Rec. Accept**	%	**Rec. Crash**	%
Jsyn	8	8	100	0	0	6	6	100	0	0
OpenImaJ	16	10	62.5	0	0	8	8	100	0	0
Lucene	9	7	77.7	2	22.2	6	5	83.3	1	16.6
Smile	12	10	83.3	0	0	5	5	100	0	0
Total	45	35	78	2	4	26	24	92	1	2

many among these loops actually improved performance and energy efficiency. Table 3.3 shows for how many loops accuracy looses were acceptable. It also shows the amount of crashes caused by approximate loops. Column 'Accept' shows the number of loops with acceptable performance, while 'Rec. Accept' indicates how many recommended loops reduced the accuracy of the application to acceptable levels. Column 'Crash' show how many loops crashed the application and 'Rec. Crash' how many recommended loops made the application crash. This data tell us that most of the loops approved by the policy have also acceptable accuracy losses.

3.4.1 RQ1: Can Approximate Unrolling Provide Satisfactory Accuracy/Energy-Performance Trade-offs?

In this section we evaluate whether Approximate Unrolling is able to provide a satisfactory trade-off for the loops it approximates.

We defined the trade-off metric T in Section 3.3.3, Equation 3.1. Trade-off is considered good when $T(P, L, E) > 1$, in other words, when the loops gains performance and energy efficiency while reducing the application's accuracy only to acceptable levels. Table 3.4 shows the amount of loops in our dataset separated by projects. Column 'Good' show for how many loops the trade-off was good, while column 'Rec. Good' indicates the number of loops recommended by the policy that improved the trade-off.

The key insight we gain from the figures the table 3.6, is that Approximate Unrolling can improve the speed-energy/accuracy trade-off in 92% of the cases in our dataset.

Table 3.4: Trade-off results for the loops of our dataset

Project	**Loops**	Good	%	Rec. Loops	Good	%
Jsyn	8	7	87.5	7	7	100
OpenImaJ	16	7	43.8	8	7	75
Lucene	9	5	55.6	6	6	100
Smile	12	5	41.7	5	5	100
Total	45	24	53.3	26	24	**92**

3.4.1.1 Policy's Quality

Approximate Unrolling is a combination of a policy and approximation strategies. In order to better grasp the policy's role in result's quality, Tables 3.2, 3.3 and 3.4 are divided into two segments. The left segment shows the improvements without any policy, while the right segment shows a full fledge Approximate Unrolling working with all its components. This shows that for a random loop in our dataset there is a 53.3% chance of trade-off improvement. However, if the policy recommends the loop, there is a 92% chance of gain.

The policy indicated two false positives (O1 and O10) that failed to gain performance in our dataset. This was due to Approximate Unrolling's C2 implementation inhibiting Loop Vectorization in later phases of the compiler. The policy had a 96% recall, this means that only one case was not recommended and still gained speed (loop J6). This occurs since the current implementation is conservative w.r.t. branching and always assume the worst case.

Answer to RQ1: Approximate Unrolling is able to provide a satisfactory trade-off for the 92% of the loops it approximates.

3.4.2 RQ2: Approximate Unrolling vs. Loop Perforation

Loop Perforation [SDMHR11] is a state-of-the-art approximate loop optimization. Hence, we compared our optimization with it. The comparison is done using our Loop Perforation's implementation on the C2 compiler. We compare both optimizations in four key aspects: scope of application, speedup, energy consumption and accuracy.

Perforation and Approximate Unrolling differs in the transformation made to the code. The key difference is that Approximate Unrolling replaces instructions in the approximate iteration, while Perforation skips it completely. If a loop is to iterate 50 times, it will iterate 50 times with Approximate Unrolling and only 25 times (or less) with Perforation. This means that Perforation does not even preserve the loop-carried dependencies. Our intuition is that for the loops it processes, Approximate Unrolling is less aggressive and preserves accuracy better.

3.4.2.1 Scope of application

Given the different transformation made by both optimizations to loops, our intuition is that they are best applied in different scenarios. According to previous works in the subject [SDMHR11], Perforation works better in patterns that can completely skip some task, like Monte-Carlo simulations, computations iteratively improving an already obtained result or data enumerations that filter or select elements in a given search space. On the other hand, we believe Approximate Unrolling will obtain best results in loops mapping values to arrays and will work well even if no previous value was mapped before. By construction, it should behave better than Loop Perforation in situations when no value of the array can be left undefined, such as sets of random numbers that follow a distribution. Another situation where Approximate Unrolling should work best is in smooth data that allow some level of granularity, such as sound, video, or word positions in a text. In these situations, Approximate Unrolling should preserve accuracy better while still improving speed. Our dataset is composed mainly of loops with these purposes, so we expect to obtain better accuracy results with Approximate Unrolling than with Perforation.

3.4.2.2 Performance, Energy, Accuracy & Trade-Off

We optimized all the loops in our dataset using both optimizations to find out which optimization produced the best results in terms of speedups, energy usage, and accuracy. Table 3.5 shows the results of this experiment. The table tell us that out of the 45 loops in our dataset, the accuracy reductions using Approximate Unrolling was smaller in 29 loops vs. being smaller in only nine using perforation. In seven loops there was no significant difference between the two in accuracy. However, Approximate Unrolling was able to provide a better performance for 14 of them, while Loop Perforation did so for 31 of the set's loops. A similar result was obtained when comparing energy (16 vs 29).

Table 3.5: Approximate Unrolling Vs Perforation

Parameter	Approx. Unroll.	Tie	Perforation
Better Accuracy	29	7	9
Higher speedup	14	0	31
Lower Energy consumption	16	0	29

Figure 3.13 shows that the accuracy obtained using Approximate Unrolling in the Musical Synthesizer was always higher. This is due to the continuous and smooth nature of the sound signal, which make it perfect for our optimization. As Approximate Unrolling does not remove loop carried dependencies, it does not accumulate error. On the other hand, error accumulation with Perforation caused the frequencies to drop,

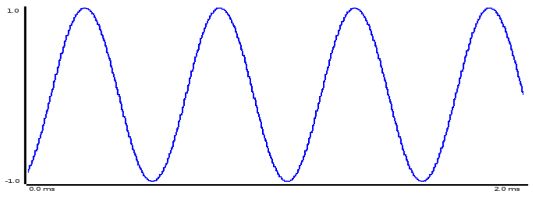


Figure 3.21: Approximation using Nearest Neighbor.

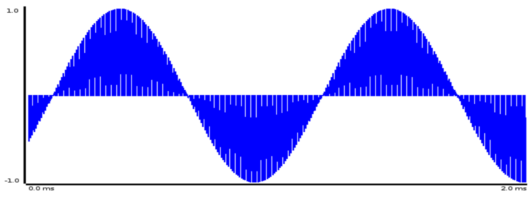


Figure 3.22: Approximation using Perforation. Notice the noise added. Also the frequency change, which causes the user to perceive a different musical note than the one expected.

changing the musical note being played. The Perforation strategy also introduced noise. The effect of both techniques on the sound can be visualized in Figures 3.21 and 3.22.

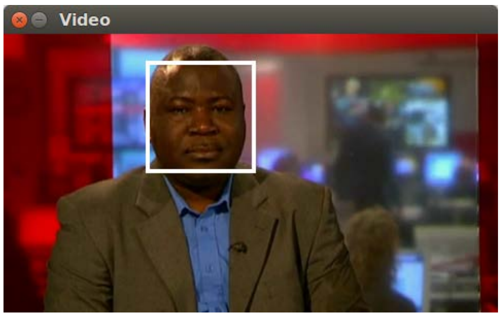


Figure 3.23: Approximation using Nearest Neighbor. The face recognition system is still able to detect the face.

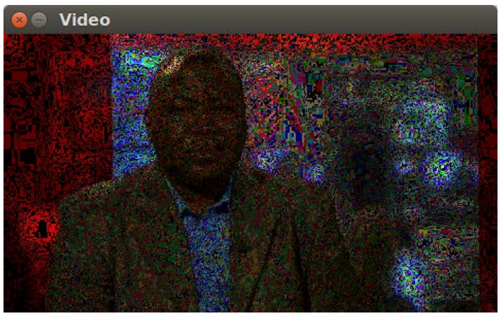


Figure 3.24: Using Perforation. The image is too noisy image and the algorithm fails to detect the face.

Figures 3.23 and 3.24 that represents the results of applying Nearest Neighbor and Perforation to a Color Lookup Table in OpenImaJ. The pictures show that the recognition algorithm fails when using perforation as consequence of the noisy image. Notice that while 12 out of 15 loops in OpenImaJ lose less accuracy using Approximate Unrolling, three loops (O5, O9, and O11) were in fact more accurate using Perforation. The reason for this is that these loops were updating already good results, a situation for which Perforation is known to work well [SDMHR11].

In the Lucene use case, Perforation crashed the application running our workload five times, while Approximate Unrolling only crashed it twice. Every time Approximate Unrolling crashed the application, so did Perforation. With the Classifier something similar happened, perforation reduced the accuracy to zero twice, while Approximate Unrolling did it only once. Again, every time Approximate Unrolling reduced recall to zero, so did Perforation.

We were expecting Perforation to be always faster, but is not the case. The performance charts of figures 3.9, 3.10, 3.11, 3.12 show that in seven loops the linear interpolation is faster than perforation. The reason is that linearly interpolated loops undergo the two more optimizations: Loop Fission and Loop Vectorization. This means that the C2 compiler splits these loop in two and then performs vectorization which uses SIMD. Also, we notice that Nearest Neighbor can have sometimes the same performance as Loop Perforation. This is because in the architecture we were running our experiments Nearest Neighbor introduces only one assembler instruction and when there is no loop carried dependencies in a loop, all instructions become dead code after the interpolation and can be removed. These results in NN adding only one assembler instruction more per iteration. These factors allowed Nearest Neighbor to match Loop's Peroration performance in 10 cases.

Answer to RQ3: Both optimizations are complementary. However, in the cases where both optimizations can be applied, Approximate Unrolling provides better accuracy.

3.4.3 RQ3: Impact of approximation ratio

Approximate Unrolling is able to use different ratios to approximate a loop. So far we have presented our results using the 1/2 ratio, i.e. we approximate one out two iterations. Other ratios are possible and in our experiments we also tried three out of four(3/4) and one out of four 1/4 approximate ratios.

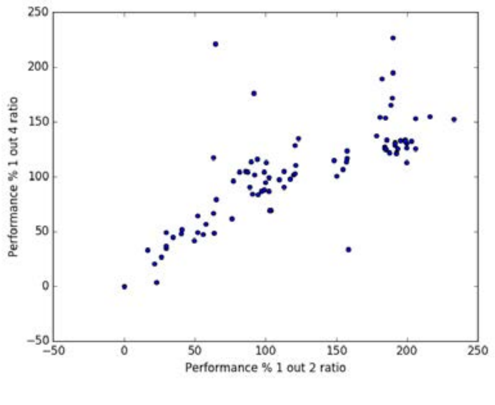


Figure 3.25: Each point represents the loop's speedup percentage. Axis X shows the improvement percentage using the 1/2 ratio, while axis Y represents the improvement using the 1/4.

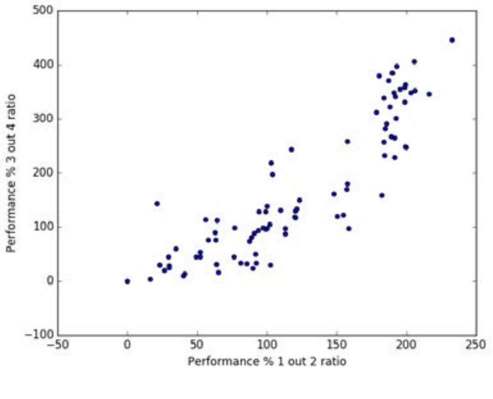


Figure 3.26: Each point represents the loop's speedup percentage. Axis X shows the improvement percentage using the 1/2 ratio, while axis Y represents the improvement using the 3/ 4.

Table 3.6: Reasons for bad Trade-off

Project	**Loops**	$\lvert NN\rvert$	$\lvert LI\rvert$	$\lvert NN \cup LI\rvert$	Bad Perf. NN	Bad Perf. LI	Bad Acc. NN	Bad Acc. LI
Jsyn	8	5	7	7	1	1	0	0
OpenImaJ	16	7	4	7	5	12	3	5
Lucene	9	5	5	5	3	3	2	2
Smile	12	5	5	5	4	6	2	1
Total	45	23	21	24	13	21	7	8

We have learned from our observations that the approximation ratio's impact in performance is roughly proportional to the number of approximate iterations. For example, if by using a 1/2 ratio we doubled performance, we can expect roughly a 300% improvement in performance using the 3/4 and a 33% improvement using the 1/4 ratio. The opposite also applies, if a loop loses performance with the 1/2 ratio, it will be even slower as we increase the ratio of approximate iterations. Figures 3.25 and 3.26 shows the relationship between different ratios. We observed similar results with energy consumption.

Accuracy-wise, we learned that results are more dependent on the context and purpose of the loop and therefore is difficult to generalize conclusions. In our Musical Synthesizer the accuracy was indeed proportional to the amount of approximation. This results were similar for the Face Recognition use case, with the exception of the loops that were updating already good results, which had mixed behaviors. In the Text Search, many loops were not dealing with signal-like data, therefore the accuracy is more dependent on strategy than the ratio, having Nearest Neighbor gives better results that Linear. In the Classifier use case, some loops were extremely forgiving and had no impact on the classifiers' recall, while other did not allow for approximation at all.

Answer to RQ4: The effect in speed and energy in our dataset is proportional to the ratio. On signal-like data, this is also true for accuracy. In other applications, accuracy reductions will depend more on the purpose and context of the loop than approximation's ratio.

3.4.4 RQ4: Loops that are the best fit for Approximate Unrolling

We have evaluated Approximate Unrolling with libraries from several domains of application, showing that Approximate Unrolling can be exploited successfully in many situations. However, not all loops in our dataset responded equally well to approximation. Table 3.6 summarizes our findings regarding the type of loop that gains the most performance with Approximate Unrolling while reducing accuracy only to acceptable levels. Columns 'NN' and 'LI' shows the number of loops in our dataset for which the trade-off is good when using Nearest Neighbor and Linear strategies respectively. Column '$\lvert NN \cup LI\rvert$' indicates how many loops improve the trade-off for at least one strategy. The Table 3.6 also shows the amount of loops fo which the trade-off is not good because they fail to improve performance (‘Bad Perf NN’ & ‘Bad Perf LI’) and the number of those who fail because accuracy losses are not acceptable (‘Bad Acc. NN’ & ‘Bad Acc. LI’).

The results indicate that while Approximate Unrolling improves the performance-energy/accuracy trade-off in 24 loops out of 45, it still misses 22. A good trade-off occurs when there is performance gains and accuracy losses are kept low.

By analyzing our dataset we have learned that the patterns allowing a performance gain are:

- Approximate Unrolling removes more instructions than the ones it inserts. Performance improvements in our technique come from removing instructions and inserting others that perform less operations. Listing 3.9 shows a loop that will gain performance when approximated. The inner ‘`for`’ loop will be completely removed from the approximate iteration and comparatively, the instructions inserted will be negligible. However, it might also occur that the code removed is less expensive than the interpolations inserted. This can happen in loops copying values from one array to another or loops initializing arrays with constant values. For example, listing 3.8 show a loop performing only constant assignments and when Approximate Unrolling replace these assignments with interpolations, performance drops occurs.

  It can also occur that the loop has many loop-carried dependencies in its body. Approximate Unrolling does not remove these to avoid error accumulation. This causes the approximate iteration to execute a lot of instructions and gain little performance.

```
for (i = 49; (i--) > 0;) {
jjrounds[i] = -2147483648;
}
```

Listing 3.8: An initialization loop from our dataset. Belongs to Apache Lucene and is used in the Text Search use case. This kind of loops does not work well with Approximate Unrolling.

```
for (int i = 0; i < N; i++) {
  float sum = 0.0F;
  for (int j = 0, jj = kernel.
      length - 1;
    j < kernel.length; j++ , jj
       --)
    sum += buffer[i + j]) *
       kernel[jj];
  buffer[i] = sum;}
```

Listing 3.9: The outter loop belongs to OpenImaJ and is used by the Face Detection use case. This loop works very well with Approximate Unrolling. The body is computationally intensive and it has no loop-carried dependencies

- There is no phase-ordering problems. The current implementation of Approximate Unrolling change the shape of loops amenable for Loop Vectorization, potentially inhibiting the Vectorization's implementation from recognizing the loop. Loop Vectorization is a machine-dependent optimization that takes advantage of the SIMD architecture present in multiple processors today. SIMD instructions yields high performance gains, which are lost if the Loop Vectorization's implementation in the C2 fails to recognize the loop as vectorizable. This is a typical phase-ordering problem. The issue is implementation-specific and we will fix in future work. Approximate Unrolling can be made to cooperate with Loop Vectorization using interleaved SIMD instructions. Loop 3.10 shows an example of a vectorizable loop that loses performance when using the current implementation of Approximate Unrolling.

```
for (int l = 0; l < (k); l++)
  falseCount[l] = count[l]) -
      (trueCount[l];
```

Listing 3.10: Example of vectorizable loop. This loops loses performance when optimized with our current implementation
of Approximate Unrolling since the C2 cannot recognize the loop as vectorizable anymore.

```
for (int i = 0; i < (imp.
    length); i++)
  importance[i] += imp[i];
```

Listing 3.11: This loop updates the weak learner's weights in the AdaBoost algorithm. The weights differs little, allowing to approximate them with nearby values. This is an example of loops working with data other than sound and images where nearby values are similar.

Performance is not the only factor required for a good trade-off. We also analyzed our dataset to discover patterns were accuracy losses remain acceptable and found the following:

- The data processed is a locally smooth function. Approximate Unrolling works well with loops dealing with arrays where nearby values are similar and small imprecisions can be tolerated. This happens naturally in signal-like data such as sound and images. However, by analyzing our dataset, we have noticed that other types of data also behaves this way. For example, the loop of Listing 3.11 updates the weights of the weak learners in the Adaptive Boost algorithm. In our workload, the differences in the weights are small (in many cases non-existent), which allows for approximation with good results. Another example is loops mapping array slots to the position of words in document search. Adjacent array slots contain contiguous words position in a document, which differ little. Users can usually tolerate a small offset when pointed the words' position. In our case, as we were looking only for documents containing the terms, it did not affect accuracy.

- The loop belongs to a forgiving zone. Other cases where the accuracy losses remain acceptable are those for loops located in zones known to be forgiving since previous studies, such as hash functions and random number generators.

Accuracy behavior varies from application to application. For the Musical Synthesizer, the accuracy loss is very much acceptable for each loop. We believe there are three reasons for this good behavior (i) the signal generators (sine wave, triangle, square and sawtooth) are locally smooth functions (ii) Approximate Unrolling does not remove loop carried dependencies and there is no error accumulation (iii) the effect of accuracy loss in the synthesizer is a harmonic distortion that is attenuated later by the synthesizer's filters. In the Face Recognition application, we also see good results that we attribute to the algorithms ability to tolerate error such as noisy or low-quality images. The loops in the Text Search application followed an interesting behavior: they were completely forgiving (i.e. has no impact in the search' accuracy) or they crashed. Loops in the Classifier followed a similar pattern: most were also completely forgiving or reduced the classifier recall almost down to 0%. In this application however, no approximate loop crashed the application. These discoveries allows us to determine the situations in which Approximate Unrolling provides a satisfactory trade-off.

Answer to RQ4: Approximate Unrolling provides a good trade-off in loops with computational intensive bodies that processes locally smooth data or that are located in forgiving zones.

3.5 Conclusions & Future Work

In this chapter we have described Approximate Unrolling, a transformation that approximates the computation inside certain loops. We formally described the shape of the loops selected for Approximate Unrolling, as well as a cost function to determine the opportunity for approximation and the transformations it performs to trade accuracy for execution time and energy consumption reductions. We have also proposed an implementation for OpenJDK Hotspot C2 Server compiler. The empirical assessment of this transformation demonstrated the ability of Approximate Unrolling to effectively trade accuracy for resource gains. We also showed that the smoother approach of Approximate Unrolling compared to Loop Perforation supports a more balanced trade-off between accuracy and performance. We learned that not all loops respond equally well to the approximation and we gained some insights on the causes for this. Hence, our future work will consist in including these findings into the optimization, improving the detection process using a cost function that favors loops whose bodies (i) contains more instructions than the ones introduced by Approximate Unrolling (ii) have a minimal number of instructions depending on a value calculated in a previous iteration (iii) represents an smooth function. Then, the selection will filter out those loops below a given threshold of the cost function.

Chapter 4

Automatic Microbenchmark Generation

Most authors in the Approximate Computing field uses benchmarks such as SPEC, SciMark and PARSEC to evaluate the speedup gains obtained with their techniques. Benchmarks are the easiest way to test performance. They are already built; they are maintained and revised by performance specialists, allows to repeat experiments easier. This is why benchmark tests are widely accepted by the scientific community. Yet, as mentioned in Chapter 2, benchmarks are large programs with thousands or millions of lines of code. Therefore, when optimizations are applied to parts of those programs, the perceived speedup in performance highly depends on the optimized zone's impact on the system, rather than the effect of the technique itself.

To investigate a techniques' impact independently of a system, performance engineers in industry have proposed a different (and complimentary) technique called Microbenchmarks. This type of performance tests allow for the finest grain performance testing (e.g., test the performance of a single loop or a variable assignment). This kind of test has been consistently used by developers in highly dependable areas such as operating systems [RCS$^+$11, JBLF10], virtual machines [Cli10], data structures [SV15], databases [LGI09], and more recently in computer graphics [NZMD15] and high performance computing[RRV$^+$13]. However, the development of microbenchmarks is still very much a craft that only a few experts master [Cli10]. In particular, the lack of tool support prevents a wider adoption of microbenchmarking.

Microbenchmarking consists in identifying a code segment that is critical for performance, a.k.a segment under analysis (SUA from now on), wrapping this segment in an independent tiny program (the payload) and having the segment executed a large number of times by the scaffold in order to evaluate its execution time. The amount of technical knowledge needed to design both the scaffold and the payload hinder engineers from effectively exploiting microbenchmarks [Ale14a, Ale14b, Cli10]. Recent frameworks such as JMH [Ale14a, Jul14b, HLST15] address the generation of the scaffold. Yet, the construction of the payload is still an extremely challenging craft.

Engineers who design microbenchmark payloads very commonly make two mistakes: they forget to design the payload in a way that prevents the JIT from performing dead code elimination [Cli10, Jul14b, Bri05, Ale14b] and Constant Folds/Propagations (CF/CP) [Ale13a, Jul14b]. Consequently, the payload runs under different optimizations than the original segment and the time measured does not reflect the time the SUA will take in the larger application. For example, Click [Cli10] found dead code in the CaffeineMark and ScifiMark benchmarks, resulting in infinite speed up of the test. Ponge also described [Jul14a] how the design of a popular set of microbenchmarks that compare JSON engines[1] was prone to "over-optimization" through dead code elimination and CF/CP. In addition to these common mistakes, there are other pitfalls for payload design, such as choosing irrelevant initialization values or reaching an unrealistic steady state.

In this work, we propose to automatically generate payloads for Java microbenchmarks, starting from a specific segment inside a Java application. The generated payloads are guaranteed to be free of dead code and prevent CF/CP. Our technique statically slices the application to automatically extract the SUA and all its dependencies in a compilable payload. Second, we generate additional code to: (i) prevent the JIT from "over-optimizing" the payload using dead code elimination (DCE) and constand folding/constant propagation(CF/CP), (ii) initialize payload's input with relevant values and (iii) keep the payload in steady state. We run a novel transformation, called sink maximization, to prevent Dead Code Elimination. We turn some SUA's local variables into fields in the payload, to mitigate (CF/CP). Finally, we maintain the payload in stable state by smart reseting variables to their initial value.

We have implemented the whole approach in a tool called AutoJMH. Starting from code segment identified with a specific annotation, it automatically generates a payload for the Java Microbenchmark Harness (JMH). JMH is the de-facto standard for microbenchmarking. It addresses the common pitfalls when building scaffolds such as Loop Hoisting and Strength Reduction, optimizations that can make the JIT reduce the number of times the payload is executed.

We run AutoJMH on the 6 028 loops present in 5 mature Java projects, to assess its ability to generate payloads out of large real-world programs. Our technique extracts 4705 SUA into microbenchmarks (74% of all loops) and correctly generates complete payloads for 3462 (60% of the loops). We also evaluate the quality of the generated microbenchmarks. We use AutoJMH to re-generate 23 microbenchmarks handwritten by performance experts. Automatically generated microbenchmarks measure the same times as the microbenchmarks written by the JMH experts. Finally, we ask 6 professional Java engineers to build microbenchmarks. All these benchmarks result in distorted measurements due to naive decisions when designing benchmarks, while AutoJMH prevents all these mistakes by construction.

To sum up, the contributions of the chapter are:

- A static analysis to automatically extract a code segment and all its dependencies
- Code generation strategies that prevent artificial runtime optimizations when running the microbenchmark
- An empirical evaluation of the generated microbenchmarks
- A publicly available tool and dataset to replicate all our experiments [2]

In section 4.1 we discuss and illustrate the challenges for microbenchmark design, which motivate our contribution. In section 4.2 we introduce our technical contribution for the automatic generation of microbenchmarks in Java. In section 4.3 we present a

[1] `https://github.com/bura/json-benchmarks`

[2] https://github.com/autojmh

qualitative and quantitative evaluation of our tool and discuss the results. Section 4.4 outlines the related work and section 4.5 concludes.

4.1 Payload Challenges

In this section, we elaborate on some of the challenges that software engineers face when designing payloads. These challenges form the core motivation for our work. In this work we use the Java Microbenchmark Harness (JMH) as to generate scaffolds. This allows us to focus on payload generation and to reuse existing efforts from the community in order to build an efficient scaffold.

4.1.1 Dead Code Elimination

Dead Code Elimination (DCE) is one of the most common optimizations engineers fail to detect in their microbenchmarks [Cli10, Jul14a, Bri05, Jul14b]. During the design of microbenchmarks, engineers extract the segment they want to test, but usually leave out the code consuming the segment's computations (the sink), allowing the JIT to apply DCE. It is not always easy to detect dead code and it has been found in popular benchmarks [Cli10, Jul14a]. For example, listing 4.1 displays a microbenchmark where the call to `Math.log` is dead code, while the call to `m.put` is not. The reason is that `m.put` modifies a public field, but the results of the `Math.log` are not consumed afterwards. Consequently the JIT will apply DCE when running the microbenchmark, which will distort the time measured.

```
Map<String, Double> m = MapUtils.buildRandomMap();
@Benchmark
public void hiddenDCE() {
  Math.log(m.put("Ten", 10));
}
```

Listing 4.1: An example of dead code

A key feature of the technique we propose in this work is to automatically analyze the mircrobenchmark in order to generate code that will prevent the JIT from running DCE on this kind of benchmark.

4.1.2 Constant Folding / Constant Propagation

In a microbenchmarks more variables has to be explicitly initialized than in the program. A quick, naive solution is to initialize these fields using constants, allowing the compiler to use Constant Folding and Constant Propagation (CF/CP) to remove computations that can be inferred. While mostly considered prejudicial for measurements, in some punctual cases a clever engineer may want to actually pass a constant to a method in a microbenchmark to see if CF/CP kicks in, since it is good for performance that a method can be constant folded. However, when not expected the optimizations causes microbenchmarks to return deceitfully good performance times.

Good examples of both DCE and CF/CP optimizations and their impact on the measurements can be found in literature [Jul14b]. Concrete evidence can also be found in the JMH examples repository[3].

4.1.3 Non-representative Data

Another source of errors when designing payloads is to run a microbenchmark with data not representing the actual conditions in which the system being measured works.

For example, suppose a maintenance being done over an old Java project and that different sort methods are being compared to improve performance, one of them being the `Collections.sort` method. Suppose that the system consistently uses `Vector<T>` but the engineer fails to see this and uses `LinkedLis<T>` in the benchmarks, concluding that `Collections.sort` is faster when given as input an already sorted list. However, as the system uses `Vector` lists, the actual case in production is the opposite: sorted lists will result in longer execution times, as shown in table 4.1, making the conclusions drawn from the benchmark useless.

Table 4.1: Execution times of *Collections.sort*

	Using a sorted list	Using an unsorted list
LinkedList	203 ns	453 ns
Vector	1639 ns	645 ns

4.1.4 Reaching Wrong Stable State

The microbenchmark scaffold executes the payload many times, warming up the code until it reaches a stable state and is not optimized anymore. A usual pitfall is to build microbenchmarks that reach stable state in conditions unexpected by the engineer. For example, if we were to observe the execution time of the Collection.sort while sorting a list, one could build the following wrong microbenchmark:

```
LinkedList<Double> m =  ListUtils.buildRandomList();
@Benchmark
public void doSort() {
  Collections.sort(m); }
```

Listing 4.2: Sorting a sorted list in each run

Unfortunately, after the first execution the list gets sorted. In consecutive executions, the list is already sorted and consequently, we end up measuring the performance of sorting an already sorted list, which is not the situation we initially wanted to measure.

[3]`http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/`

4.2 AutoJMH

AutoJMH automatically extracts a code segment and generates a complete payload with inputs that reflect the behavior of the segment in the original application. The generation process not only wraps the segment in an independent program, it also mitigates the risks of unexpected DCE and CF/CP optimizations and ensures that it will reach stable state in the same state executed by the SUA during the unit tests.

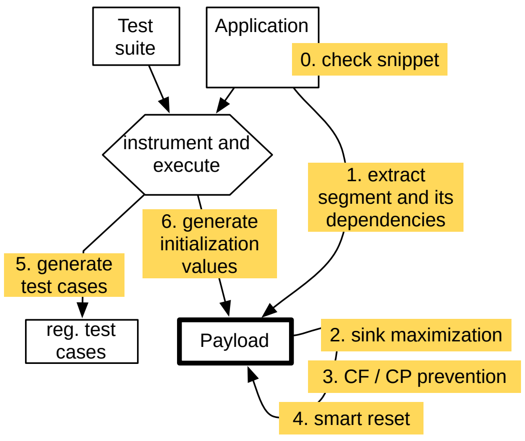


Figure 4.1: Global process of AutoJMH for payload generation.

Figure 4.1 illustrates the different steps of this process. If the SUA satisfies a set of preconditions (detailed in section 4.2.1), AutoJMH extracts the segment into a wrapper method. Then, the payload is refined to prevent dead code elimination, constant folding and constant propagation (steps 2, 3), as well as unintended stable state (step 4) when the payload is executed many times. The last steps consist in running the test suite on the original program to produce two additional elements: a set of data inputs to initialize variables in the payload; a set of regression tests that ensure that the segment has the same functional behavior in the payload and in the original application.

In the rest of this section we go into the details of each step. We illustrate the process through the creation of a microbenchmark for the `return` statement inside the `EnumeratedDistribution::value()` method of Apache Common Math, shown in listing 4.3. The listing also illustrates that a user identifies a SUA by placing the Javadoc-like comment `@bench-this` on top of it. This comment is specific to AutoJMH and can be put on top of every statement. The resulting payload is shown listing 4.4.

```
double value(double x, double... param) throws
DimensionMismatchException, NullArgumentException {
  validateParameters(param);
  /** @bench-this */
    return Sigmoid.value(x, param[0], param[1]);
  }
```

Listing 4.3: An illustrating example: a SUA in commons.math

In listing 4.4 we can see that AutoJMH has wrapped the `return` statement into a method annotated with `@Benchmark`. This annotation is used to indicate the wrapper method that is going to be executed many times by the JMH scaffold. The private static method `Sigmoid.value` has been extracted also into the payload, since it is needed by the SUA. AutoJMH has turned variables `x` and `params` into fields and provides initialization code from them, loading values from a file, which is part of our strategy to avoid CF/CP. Finally, AutoJMH ensures that some value is returned in the wrapper method to avoid DCE.

```
class MyBenchmark {
double[] params;    double x;
@Setup
void setup() {
  Loader l = new Loader("/data/Sigmoid_160.dat");
  x = l.loaddouble();
  params = l.loaddoubleArray1();
}
double Sigmoid_value(
  double x, double lo, double hi) {
    return lo + (hi - lo) / (1 + FastMath.exp(-x));
}
@Benchmark
public double payloadWrapper() {
  return Sigmoid_value(x, params[0], params[1])}
}
```

Listing 4.4: An illustrating example: the payload generated by AutoJMH

4.2.1 Preconditions

The segment extraction is based on a static analysis and focuses on SUAs that meet the following conditions. These preconditions ensure that the payload can reproduce the same conditions than those in which the SUA is executed in the original program.

1. Initialized variables used by the SUA are of the following types: primitive (`int, double, boolean`), their class counterparts (`Integer, Double, Boolean`), String, types implementing the Serializable interface, or, collections and arrays of all the above. Non-initialized variables used by the SUA can be of any public type. This condition ensures that AutoJMH can store the values of all variables used by the SUA
2. None of the methods invoked inside the SUA can have a target not supported in item 1. This ensures that AutoJMH is able to extract all methods used by the SUA.
3. All private or protected methods used by the SUA can be resolved statically. Dynamically resolved methods have a different performance behavior than statically resolved ones [Ale15]. Using dynamic slicing we could make available to the microbenchmark a non-public dynamic method, but we would distort its performance behavior.

4. The call graph of all methods used by the SUA cannot be more than a user-defined number of levels deep before reaching a point in which all used methods are public. This sets a stopping criterion for the exploration of the call graph.

4.2.2 SUA extraction

AutoJMH starts by extracting the segment under analysis (SUA) to create a compilable payload using a custom forward slicing method over the Abstract Syntax Tree (AST) of the large application, which includes the source code of the SUA. The segment's location is marked with the `@bench-this` Javadoc-like comment, introduced by AutoJMH to select the segments to be benchmarked. If the SUA satisfies the preconditions, AutoJMH statically slices the source code of the SUA and its dependencies (methods, variables and constants) from the original application into the payload. Non-public field declarations and method bodies used by the SUA are copied to the payload, their modifiers (static, final, volatile) are preserved.

Some transformations may be needed in order to achieve a compilable payload. Non-public methods copied into the payload are modified to receive their original target in the SUA as the first parameter (e.g., `data.doSomething()` becomes `doSomething(data)`). Variable and method may be renamed to avoid name collision and to avoid serializing complex objects. For example, if a segment uses both variable `data` and a field `myObject.data`, AutoJMH declares two public fields: `data` and `myObject_data`. When method renaming is required, AutoJMH uses the fully qualified name.

At the end of the extraction phase, AutoJMH has sliced the SUA code into the payload's wrapper method. This relieves the developer from a very mechanical task and its automation reduces the risks of errors when copying and renaming pieces of code. Yet, the produced payload still needs to be refined in order to prevent the JIT from "over-optimizing" this small program.

Preserving the original performance conditions We aim at generating a payload that recreates the execution conditions of the SUA in the original application. Hence, we are conservative in our preconditions before slicing. We also performed extensive testing to be sure that the code modifications explained above do not distort the original performance of the SUA. These tests are publicly available [4]. Then, all the additional code generated by AutoJMH to avoid DCE, initialize values, mitigate CF/CP and keep stable state, is inserted before or after the wrapped SUA.

4.2.3 Preventing DCE with Sink Maximization

During the extraction of the SUA, we may leave out the code consuming its computations (the sink), giving the JIT an opportunity for dead code elimination (DCE), which would distort the time measurement. AutoJMH handles this potential problem featuring a novel transformation that we call Sink maximization. The transformation

[4]`https://github.com/autojmh/syntmod`

appends code to the payload, which consumes the computations. This is done to maximize the number of computations consumed while minimizing the performance impact in the resulting payload.

There are three possible strategies to consume the results inside the payload:

- Make the payload wrapper method return a result. This is a safe and time efficient way of preventing DCE, but not always applicable (e.g., when the SUA returns void).
- Store the result in a public field. This is a time efficient way of consuming a value, yet less safe than the previous solution. For example, two consecutive writes to the same field can make the first write to be marked as dead code. It can also happen that the payload will read from the public field with a new value, modifying its state.
- JMH Black hole methods. This is the safest solution, which does not modify the microbenchmark's state. Black holes (BH) are methods provided by JMH to make the JIT believe their parameters are used, therefore preventing DCE. Yet, black holes have a small impact on performance.

A naive solution is to consume all local variables live at the end of the method with BHs. Yet, the accumulation of BH method calls can be a considerable overhead when the execution time of the payload is small. Therefore, we first use the `return` statement at the end of the method, taking into consideration that values stored in fields are already sinked and therefore do not need to be consumed. Then, we look for the minimal set of variables covering the whole sink of the payload to minimize the number of BH methods needed.

Sink maximization performs the following steps to generate the sink code:

1. Determine if it is possible to use a `return` statement.
2. Determine the minimal set of variables V_{min} covering the sink of the SUA.
3. When the use of `return` is possible, consume one variable from V_{min} using one `return` and use BHs for the rest. If no `return` is possible, use BHs to consume all local variables in V_{min}.
4. If a return is required to satisfy that all branches return a value and there is no variables left in V_{min}, return a field.

To determine the minimal set V_{min}, the AutoJMH converts the SUA code into static single assignment (SSA) form[TC11] and builds a value dependency graph (VDG) [WCES94]. In the VDG, nodes represent variables and edges represent direct value dependencies between variables. For example, if the value of variable A directly depends on B, there is an edge from B to A. An edge going from one variable node to a *phi* node merging two values of the same variable is a back-edge. In this graph, sink-nodes are nodes without ingoing edges.

Initially, we put all nodes of the VDG in V_{min}, except those representing fields values. Then, we remove all variables that can be reached from sink-nodes from V_{min}.

After doing this, if there are still variables in V_{min} other than the ones represented by sink-nodes, we remove the back-edges and repeat the process.

```
int d = 0; a = b + c;
if ( a > 0 ) {
    d = a + h;
    a = 0;
}
b = a;
```

Listing 4.5: A few lines of code to exemplify Sink maximization

To exemplify the process of finding V_{min} within Sink Maximization let us consider listing 4.5. The resulting VDG graph is represented in figure 4.2. Sink nodes are nodes d and b1, which are represented as rounded nodes. The links go from variables to their dependencies. For example, d depends on a0 and h. Since it is not possible to arrive to all nodes from a single sink d or b1, in the example $V_{min} = \{d, b1\}$. Consequently both d and b must be consumed in the payload.

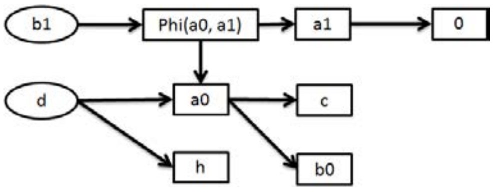


Figure 4.2: VDG of listing 4.5

4.2.4 CF/CP Mitigation

Since all SUA are part of a larger method, they most often use variables defined upfront in the method. These variables must be declared in the payload. Yet, naively declaring these variables might let the JIT infer the value of the variables at compile time and use constant folding to replace the variables with a constant. Meanwhile, if this was possible in the original system, it should also be possible in the payload. The challenge is then to detect when CF/CP must be avoided and when it must be allowed to declare variables and fields accordingly.

AutoJMH implements the following rules to declare and initialize a variable in the payload:

- Constants (`static final` fields) are initialized using the same literal as in the original program.
- Fields are declared as fields, keeping their modifiers (static, final, volatile) and initialized in the `@Setup` method of the microbenchmark. Their initial values are probed through dynamic analysis and logged in a file for reuse in the payload (cf. section 4.2.6 for details about this probing process).

- Local variables are declared as fields and initialized in the same way, except when (a) they are declared by assigning a constant in the original method and (b) all possible paths from the SUA to the beginning of the parent method include the variable declaration (i.e. the variable declaration dominates [TC11] the SUA) , in which case their original declaration is copied into the payload wrapper method. We determine whether the declaration of the variable dominates the SUA by analyzing the control flow graph of the parent method of the SUA.

Listing 4.4 shows how the variables `x` and `params` are turned into fields and initialized in the `@Setup` method of the payload. The `@Setup` method is executed before all the executions of the wrapper method and its computation time is not measured by the scaffold.

4.2.5 Keep Stable State with Smart Reset

In Section 4.1 we discussed the risk for the payload to reach an unintended stable state. This happens when the payload modifies the data over which it operates. For example, listing 4.6 shows that variable `sum` is auto-incremented. Eventually, `sum` will be always bigger than `randomValue` and the payload will stop to execute the `return` statement.

```
public T sample() {
  final double randomValue = random.nextDouble();
  double sum = 0;
  /** @bench-this */
  for (int i = 0; i < probabilities.length; i++) {
    sum += probabilities[i];
    if (randomValue < sum) return singletons.get(i);
  }
  return singletons.get(singletons.size() - 1);
}
```

Listing 4.6: Variable `sum` needs to be reset to stay in the same state

AutoJMH assumes that the computation performed in the first execution of the payload is the intended one. Hence, it automatically generates code that resets the data to this initial state for each run of the SUA. Yet, we implement this feature of AutoJMH carefully to bring the reset code overhead to a minimum. In particular, we reset only the variables influencing the control flow of the payload. In listing 4.7 AutoJMH determined that `sum` must be reset, and it generates the code to do so.

```
@Benchmark
public double doBenchmark() {
  sum = sum_reset; //<- SMART RESET HERE!
  for (int i = 0; i < probabilities.length; i++) {
    sum += probabilities[i];
    if (randomValue < sum) return singletons.get(i);
  }
  return sum;
}
```

Listing 4.7: Variable `sum` is reset by code appended to the microbenchmark

To determine which variables must be reset, AutoJMH reuses the VDG built to determine the sinks in the Sink maximization phase. We run Tarjan's Strongly Connected Components algorithm to locate cycles in the VDG, and all variables inside a cycle are considered as potential candidates for reset. In a second step we build a Control Flow Graph (CFG) and we traverse the VDG, trying to find paths from variables found in the branching nodes of the CFG to those found in the cycles of the VDG. All of the variables that we succesfully reach are marked for reset.

4.2.6 Retrieving Inputs for the Payload

The last part of the microbenchmark generation process consists in retrieving input values observed in the original application's execution (steps 5 and 6 of figure 4.1). To retrieve these values, we instrument the original program to log the variables just before and after the SUA. Then, we run once the test cases that cover the SUA in order to get actual values. The user may also configure the tool to use any program executing the SUA.

To make the collected values available to the payload, AutoJMH generates a specific JMH method marked with the `@Setup` annotation (which executes only once before the measurements), containing all the initialization code for the extracted variables. Listing 4.4 shows an example where the initial values of variables `x` and `params` are read from file.

```
@Test
public void testMicroBench () {
  Loader l = new Loader();
  //Get values recorded before execution
  l.openStream("/data/Sigmoid_160.dat");
  MyBenchmark m = new MyBenchmark();
  m.x = l.readdouble();
  m.params = l.readdoubleArray1();
  double mResult = m.payloadWrapper();
  //Check SUA's output is equal to payload's output
  l.openStream("/data/Sigmoid_160_after.dat");
  assertEquals(m.x, l.readdouble());
  assertArrDblEquals(m.params, l.readdoubleArray1());
  assertEquals(mResult, m.payloadWrapper());
}
```

Listing 4.8: Generated unit test to ensure that the microbenchmark has the same functional behavior than the SUA

4.2.7 Verifying Functional Behavior

To check that the wrapper method has the same functional behavior as the SUA in the original application (i.e. produces the same output given the same input), AutoJMH generates a unit test for each microbenchmark, where the outputs produced by the microbenchmark are required to be equal to the output values recorded at the output of the SUA. These tests serve to ensure that no optimization applied on the benchmark interferes with the expected functional behavior of the benchmarked code. In the test, the benchmark method is executed twice to verify that the results are consistent within two executions of the benchmark and signal any transient state. Listing 4.8 shows a unit test generated for the microbenchmark of listing 4.4.

Table 4.2: Reach of AutoJMH

PROPERTY	MATH	%	VECT	%	LANG	%	JSYN	%	Img2	%	Total	%
Total Loops	2851		1498		501		306		926		6082	
Payloads generated	2086	73	1377	92	408	81	151	49	683	74	4705	77
Payloads generated & initialized	1856	65	940	63	347	69	88	29	254	27	3485	57
Microbenchmarks generated	**1846**	**65**	**934**	**62**	**345**	**69**	**84**	**29**	**253**	**27**	**3462**	57
Rejected:	765	26	121	8	93	19	155	50	243	26	1377	23
* Variables unsupported:	601	21	81	5	53	11	123	40	169	18	1027	17
+ Unsupported type collection	52	2	12	1	2	0,4	18	6	15	2	99	2
+ Type is not public	132	5	2	0,1	8	2	23	7	0	-	165	3
+ Type is not storable	417	15	67	5	43	9	82	27	154	17	763	13
* Invocations unsupported:	164	6	40	3	40	8	32	10	74	8	350	6
+ Target unssuported	150	5	34	3	37	7,39	28	9	74	8	323	5
+ Levels too deep	0	0	0	0	2	0,4	0	0	0	0	2	0.03
+ Private constructor	3	0,1	3	0,2	0	0	3	1	0	0	9	0.1
+ Protected abstract method	11	0,4	3	0,2	1	0,2	1	0,3	0	0	16	0.3
Test failed	10	0,4	6	0,4	2	0,4	4	1,3	1	0,1	23	0.4

4.3 Evaluation

We perform a set of experiments on large Java programs to evaluate the effectiveness of our approach. The purpose of the evaluation is twofold. First, a quantitative assessment of AutoJMH aims at evaluating the scope of our program analysis, looking at how many situations AutoJMH is able to handle for automatic microbenchmark generation. Second, two qualitative assessments compare the quality of AutoJMH's generated microbenchmarks with those written by experts and with those built by expert Java developers who have little experience in microbenchmarking. We investigate these two aspects of AutoJMH through the following research questions:

RQ1: How many loops can AutoJMH automatically extract from programs into microbenchmarks?

In addition to the generation of accurate microbenchmarks, it is important to have a clear understanding of the reach of AutoJMH's analysis capacities. Remember that AutoJMH can only handle those segments that meet certain preconditions. Therefore, we need to quantify the impact of these conditions when analyzing real-world code.

RQ2: How does the quality of AutoJMH's generated microbenchmarks compare with those written by experts?

Our motivation is to embed expert knowledge into AutoJMH, to support Java developers who have little knowledge about performance evaluation and who want to get accurate microbenchmark. This research question aims at evaluating whether our technique can indeed produce microbenchmarks that are as good as the ones written by an expert.

RQ3: Does AutoJMH generate better microbenchmarks than those written by engineers without experience in microbenchmarking?

Here we want to understand to what extent AutoJMH can assist Java developers wanting to use microbenchmarks.

4.3.1 RQ1: Automatic Segment Extraction

We automatically annotate all the 6 028 loops of 5 real Java projects with the `@bench-this` annotation to find out to what extent the tool is able to automatically extract loops and generate corresponding payloads. We focus on the generation of benchmarks for loops since they are often a performance bottleneck and they stress AutoJMH's capacities to deal with transient states, although the only limitations to the slicing procedure are the ones described in section 4.2.1.

We selected the following projects for our experiments, because their authors have a special interest in performance: Apache Math is the Apache library for mathematics and statistics; Vectorz is a vector and matrix library, based around the concept of N-dimentional arrays. Apache Common Lang provides a set of utility methods to handle Java core objects; Jsyn is a well known library for the generation of music software synthesizers. ImageLib2 is the core library for the popular Java scientific image processing tool ImageJ. Exact versions of these projects can be found in AutoJMH's repository[5].

Table 4.2 sumarizes our findings, one column for each project and the last column shows totals. The row "Payloads generated" shows the number of loops that AutoJMH succesfully analyzed and extracted in a payload code. The row "Payloads Generated & Initialized" refines the previous number, indicating those payloads for which AutoJMH was able to generate code and initialization values (i.e. they were covered with at least one unit test). The row "Microbenchmarks generated" further refines the previous numbers, indicating the amount of loops for which AutoJMH was able to generate and initialize a payload that behaves functionally the same as the SUA (i.e. equal inputs produce equal results). The rows below detail the specific reason why some loops could not be extracted. We distinguish between "Variables unsupported" or "Invocations Unsupported". As we can see, the main reason for rejection are unsupported variables. Finally, row "Test Failed" shows the number of microbenchmarks that failed to pass the generated regressions tests. The percentages are overall percentages.

The key result here is that out of the 6 028 loops found in all 5 projects, AutoJMH correctly analyzed, extracted and wrapped 3 462 loops into valid microbenchmarks. These microbenchmarks resulted from 3 485 payloads for which AutoJMH was able to generate and find initialization values and who's regression test did not fail. In total,

[5]`https://github.com/autojmh/autojmh-validation-data.git`

AutoJMH generated the code for 4 705 payloads. The tool rejected 1377 loops because they did not meet the preconditions.

Looking into the details, we observe that Vectorz and Apache Lang contain relatively more loops that satisfy the preconditions. The main reason for this is that most types and classes in Vectorz are primitives and serializables, while Apache Lang extensively uses Strings and collections. Apache Math also extensively uses primitives. The worst results are to JSyn: the reason for this seems to be that the parameters to the synthesizers are objects instead of numbers, as we initially expected.

The results vary with the quality of the test suite of the original project. In all the Apache projects, almost all loops that satisfy the precondition finally turn into a microbenchmark, while only half of the loops of Vectorz and JSyn that can be processed by AutoJMH are covered by one test case at least. Consequently, many payloads cannot be initialized by AutoJMH, because it cannot perform the dynamic analysis that would provide valid initializations.

```
outer:
for (int i = 0; i < csLen; i++) {
  final char ch = cs.charAt(i);
  /** @bench-this */
  for (int j = 0; j < searchLen; j++) {
    if (searchChars[j] == ch) {
      if (i < csLast && j < searchLast && Character.isHighSurrogate(ch))
          {
        if (searchChars[j + 1] == cs.charAt(i + 1)) {
          continue outer; }
      } else { continue outer; }}}
```

Listing 4.9: The SUA depends on outer code to work properly

Table 4.2 also shows that some microbenchmarks fail regression tests. A good example is the inner loop of listing 4.9, extracted from Apache Common Lang. This loop depends on the `ch` variable, obtained in its outer loop. In this case, AutoJMH generates a payload that compiles and can run, but that does not integrate the outer loop. So the payload's behavior is different from the SUA and the regression tests fails.

It is worth mentioning that while AutoJMH failed to generate the inner loop, it did generate a microbenchmark for the outer one.

Answer to RQ1: AutoJMH was able to generate 3 485 microbenchmarks out of 6 028 loops found in real-word Java programs, and only 23% of the analyzed loops did not satisfy the tool's preconditions.

4.3.2 RQ2:Microbenchmarks Generated by AutoJMH vs Handwritten by Experts

To answer RQ2, we automatically re-generate mircrobenchmarks that were manually designed by expert performance engineers. We assess the quality of the automatically generated microbenchmarks by checking that the times they measure are similar to the times measured by the handwritten microbenchmarks.

4.3.2.1 Microbenchmarks Dataset

We re-generate 23 JMH microbenchmarks that were used to find 8 documented performance regression bugs in projects by Oracle [6] and Sowatec AG [Dmi15]. We selected microbenchmarks from Oracle, since this company is in charge of the development of Hotspot and JMH. The flagship product of Sowatec AG, Arregulo [7], has reported great performance results using microbenchmarks. The microbenchmarks in our dataset contained several elements of Java such as conditionals, loops, method calls, fields. They where aimed at variety of purposes and met the AutoJMH preconditions.

Follows a small description of each one of the 23 microbenchmarks (MB) in our dataset:

MB 1 and 2: Measure the differences between the two methods `ArrayList.add` and `ArrayList.addAll` when adding multiple elements.

MB 3 to 5: Compare different strategies of creating objects using reflection, using as baseline the operator `new`.

MB 6: Measure the time to retrieve fields using reflection.

MB 7 to 9: Compare strategies to retrieve data from maps when the key is required to be a lower case string.

MB 10 and 11: Compare the `ConcurrentHashMap.get` method vs. the `NonBlockingHashMapLong.get` method.

MB 12 to 14: See whether `BigInteger.value` can be constant folded when given as input a number literal.

MB 15 and 16: Contrasts the performance of `Math.max` given two numbers vs. a greater than ($a > b$) comparison.

MB 17: Evaluate the performance of the `Matcher.reset` method.

MB 18 to 23: Evaluate the performance of the `String.format` method using several types of input (`double`, `long`, `String`).

4.3.2.2 Statistical Tests

We use the statistical methodology for performance evaluation introduced by George [GBE07] to determine the similarity between the times measured by the automatically generated microbenchmarks and the handwritten ones. This consists in finding the confidence interval for the series of execution times of both programs and to check whether they overlap, in which case there is no statistical reason to say they are different. We run the experiment following the recommended methodology, considering 30 virtual machine invocations, 10 of which run for microbenchmarks and 10 warm up iterations to reach steady state. We select a confidence level of 0,05.

[6]http://bugs.java.com. Bugs ids: 8152910, 8050142, 8151481 and 8146071

[7]`http://www.sowatec.com/en/solutions-services/arregulo/`

To determine whether AutoJMH actually defies the pitfalls shown in section 4.1, we also generate three other sets of 23 microbenchmarks. Each set of microbenchmark is prone to the following pitfall: DCE, CF/CP and wrong initial values. DCE was provoked by turning off sink maximization. CF/CP was provoked by inverting the rules of variable declaration where constants (static final fields) are declared as regular fields and initialized from file; fields are redeclared as constants (static final field) and initialized using literals (`10, "zero", 3.14f`); local variables are always declared as local variables and initialized using literals. In the third set, we feed random data as input to observe differences in measurements caused by using different data. Using these 3 different sets of microbenchmarks, we performed the pairwise comparison again between them and the handwritten microbenchmarks.

Table 4.3: Comparison of generated vs handwritten benchmarks

#	Set	Successful tests
1	**Generated with AUTOJMH**	**23 / 23**
2	*DCE*	0 / 23
3	*CF/CP*	11 / 23
4	*Bad initialization*	3 / 23

Table 4.3 shows the results of this experiment. Column “Successful tests" shows for how many of the 23 automatically generated microbenchmarks measured the same times as the ones written by experts. Row 1 shows the set generated with all features of AutoJMH. Rows 2, 3 and 4 the ones generated with induced errors.

4.3.2.3 Analysis of the Results

The key result of this set of experiments is that all the 23 microbenchmarks that we re-generated using all the features of AutoJMH measure times that are statistically similar to those measured by the ones handwritten by experts, while microbenchmarks with induced errors consistently drift away from this baseline. For us, this is an strong indication that AutoJMH actually defies the pitfalls of section 4.1.

Row 2 of table 4.3 shows the strong impact of DCE on the accuracy of microbenchmarks: 100% of microbenchmarks that we generate without sink maximization measure significantly different times from the times of handwritten microbenchmarks. The inverted rules for CF/CP take a toll on 12 microbenchmarks, for example the result of a comparison between two constants is also a constant (MB 15) and therefore there is no need to perform the comparison. Eleven microbenchmarks generated with wrong variable declarations still measure similar times, because some SUA cannot be constant folded (e.g., the `Map.get` method in in MB 7). Finally, line 5 shows that passing wrong initial values produces different results, since adding 5 elements to a list takes less time than adding 20 (MB 1, 2) or converting PI (3,14159265) to string is certainly slower than an integer such as 4 (MB 18 to 23). The three cases that measured correct times occur when the fields initialized in the payload are not used (as is the case in MB 5).

The code for all the microbenchmarks used in this experiment, as well as the program and the unit test used to rebuild them, can be found in the website of AutoJMH[8].

Answer to RQ2: microbenchmarks automatically generated by AutoJMH systematically perform as good as benchmarks built by a JMH experts with a confidence level of 0.05. The code generated to prevent DCE, CF/CP and initialize the payload plays a significant role in the quality of the generated microbenchmarks.

4.3.3 RQ3: AutoJMH vs Engineers without Microbenchmarking Experience

For this research question, we consider 5 code segments, all contained in a single class and we ask 6 professional Java developers with little experience in performance evaluation to build a microbenchmark for each segment. This simulates the case of software engineers looking to evaluate the performance of their code without specific experience in time measurement. This is a realistic scenario, as many engineers arrive to microbenchmarking due to an eventual need, gathering the knowledge they require by themselves using available resources as Internet tutorials and conferences.

We provided all participants a short tutorial about JMH. All participants had full access to Internet during the experiment and we individually answered all questions relative to better microbenchmarking. Participants were also reminded that code segments may have multiple performance behaviors and that otherwise noticed, they should microbenchmark all behaviors they could find.

4.3.3.1 Segments Under Analysis

Each of the 5 code segments is meant to test one different feature of AutoJMH.

SUA 1 in listing 4.10: participants were requested to evaluate the execution time of the `for` loop. Here we evaluate a segment which execution time depends on the different input's types. The parameter `c` of `addFunction` is of type `MyFunction`, which is inherited by two subclasses, both overriding the `calc` method. The calculations performed by both subclass are different, which required several microbenchmarks to evaluate all possibilities.

SUA 2 and 3 in listing 4.11: participants were requested to evaluate the time it takes to add one element into an array list, and the time it takes to sort a list of 10 elements. Here we wanted to test the participant's ability at using different reset strategies to force the microbenchmark reach stable state measuring the desired case. The payload for SUA 2 must constrain the list size , otherwise the JVM runs out of memory. For SUA 3 it is necessary to reset the list into an unordered state.

SUA 4 and 5 in listing 4.12: participants were requested to estimate how long takes the expression to execute. The segments consist of simple mathematical expressions

[8]`https://github.com/autojmh`

meant to investigate if participants are able to avoid DCE and constant folding when transplanting a SUA into a payload.

```
addFunction(
   MyFunction c) {
  if (c == null)
     c = new FunA();
  //SUA #1:
  for (int i = 0;
    i < 100; i++)
    sinSum += c.calc(
        i);}
```

Listing 4.10: SUA 1. Differents types of 'c' define performance

```
appendSorted(
    ArrayList<Integer>
     a,
  int value) {
  //SUA #2:
  a.add(value);
  //SUA #3:
  a.sort(new
      Comparator<
      Integer>() {
    compare(Integer
       o1, Integer o2
       ) {
     return o1 - o2
         ;}});}
```

Listing 4.11: Segments 2 and 3

```
//SUA #4
angle += Math.abs(
    Math.sin(y)) / PI;

//SUA #5
double c = x * y;
```

Listing 4.12: SUAs 4 and 5

All microbenchmarks used in this experiment are publicly available in the github repository of AutoJMH

4.3.3.2 Resulting Microbenchmarks

Figure 4.3 shows the execution times measured by all microbenchmarks. The y-axis shows execution times in milliseconds (log scale). On the x-axis we show 6 clusters: MB1a and MB1b for the two performance behaviors of SUA 1 and MB2 to MB5 for all other segments. Each cluster includes the time measured by the microbenchmarks designed by the 6 Java developers. In each cluster, we add two microbenchmarks: one generated by AutoJMH and one designed manually by us and that has been reviewed by the main developer of JMH. The latter microbenchmark (for short: the expert) is used as the baseline for comparison. We use the similiraty of execution times for comparison: the closest to the baseline, the better.

First, we observe that the times for the AutoJMH and the baseline microbenchmarks are consistently very close to each other. The main differences we can see are located in SUAs 2 and 3. This is because AutoJMH uses a generic reset strategy consisting in clearing the list and adding the values, which is robust and performs well in most cases. However, the expert microbenchmarks and the one made by Engineer 6 for SUA 3 featured specific reset strategies with less overhead. The best strategy to reset in SUA 2 is to reset only after several calls to the `add` method have been made, distributing the reset overhead and reducing the estimation error. In the expert benchmark for SUA 3, each element is set to a constant value. A clever trick was used by engineer 6 in SUA 3 [9]: the `sort` method was called twice with two different comparison functions

[9] https://github.com/autojmh/autojmh-validation-data/blob/master/eng6/src/main/java/fr/inria/diverse/

(with equivalent performance), changing the correct order in every call. This removes the need to reset the list, since every consecutive call to `sort` is considered unordered.

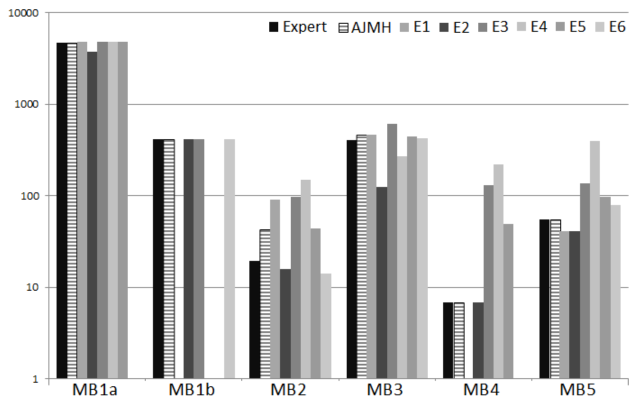


Figure 4.3: Execution times comparison between microbenchmarks generated by AutoJMH those manually built by Java developers and one JMH expert

Second, we observe that Java developers build microbenchmarks that measure times that are very different from the baseline. In order to understand the root cause of these differences, we manually review all the microbenchmark. Here we observe that the participants did encounter the pitfalls we expected for each kind of segment: 3 participants fail to distinguish 2 performance behaviors in MB1; 3 participants made mistakes when initializing MB2 and MB3; we found multiple issues in MB4 and MB5, where 3 engineers did not realize that their microbenchmark was optimized by DCE, Engineer 6 allowed parts of its microbenchmark to be constant folded and 3 participants bloated to some extend their microbenchmark with overhead. An interesting fact was that Engineer 6 was aware of constant folding, since he asked about it, meaning that a trained eye is needed to detect optimizations, even when one knows about them.

Answer to RQ3: microbenchmarks generated by AutoJMH prevent mistakes commonly made by Java developers without experience in microbenchmarking.

4.3.4 Threats to Validity

The first threat is related to the generalizablity of observations. Our qualitative evaluation was performed only with 5 segments and 6 participants. Yet, segments were designed to be as different as possible and to cover different kinds of potential pitfalls. The quantitative experiment also allowed us to test AutoJMH on a realistic code base, representative of a large number of situations that can be encountered in Java applications.

`autojmh/validation/eng6/TransientStateListSortEng6.java`

AutoJMH is a complex tool chain, which combines code instrumentation, static and dynamic analysis and code generation. We did extensive testing of our the whole infrastructure and used it to generate a large number of microbenchmarks for a significant number of different applications. However, as for any large scale experimental infrastructure, there are surely bugs in this software. We hope that they only change marginal quantitative things, and not the qualitative essence of our findings. Our infrastructure is publicly available on Github.

4.4 Related work

We are not aware of any other tool that automatically generates the payload of a microbenchmark. However, there are works related to many aspect of AutoJMH.

Performance Analysis The proper evaluation of performance is the subject of a large number of papers [GBE07, MDHS09, Ale14a, HLST15, Cli10]. They all point out non-determinism as the main barrier to obtain repeatable measurements. Sources of non-determinism arise in the data, the code [Jul14b], the compiler[MDHS09], the virtual machine[GEB08] the operating system [MDHS09] and even in the hardware[CB13]. Various tools and techniques aim at minimizing the effect of non-determinism at each level of abstraction[MDHS09, CB13, GEB08]. JMH stands at the frontier between code and the JVM by carefully studying how code triggers JVM optimizations[Ale13a]. AutoJMH is at the top of the stack, automatically generating code for the JMH payload, avoiding unwanted optimizations that may skew the measurements.

Microbenchmarking determines with high precision the execution time of a single point. This is complementary to other techniques that use profiling [SLPG15, AAFM10] and trace analysis [HDG$^+$12, GFX12] that cover larger portions of the program at the cost of reducing the measurement precision. Symbolic execution is also used to analyze performance [BJS09, ZED11] however, symbolic execution alone cannot provide execution times. Finally several existing tools are specific for one type of bug [NCRL15, ODL15] or even for one given class of software, like the one by Zhang [ZED11] which generates load test for SQL Servers.

AutoJMH is a tool that sits between profiling/trace analysis and microbenchmarking, providing execution times for many individuals points of the program with high precision.

Performance testing in isolation Specially close to our work are the approaches of Horkỳ [HLST15, HLM$^+$15], Kuperberg [KOR] and Pradel [PHG14].

Microbenchmarking, and therefore AutoJMH, evaluate performance by executing one segment of code in isolation. A simpler alternative favored by industry are performance unit tests [CB13, Dav03], which consist in measuring the time a unit test takes to run. Horkỳ et.al. proposes methodologies and tools to improve the measurements that can be obtained using performance unit tests uses, unlike AutoJMH, which uses unit tests only to collect initialization data. Kuperberg creates microbenchmarks for Java

APIs using the compiled bytecode. Finally, Pradel proposes a test generator tailored for classes with high level of concurrency, while AutoJMH uses the JMH built-in support for concurrency. All these approaches warm-up the code and recognize the intrinsic non-determinism of the executions.

The main distinctive feature of AutoJMH over these similar approaches is its unique capability to measure at the statement level. These other approaches generate test execution for whole methods at once. Baudry [BARCM15a] shows that some methods use code living as far as 13 levels deep in the call stack, which gives us an idea of how coarse can be executing a whole test method. AutoJMH is able to measure both complete methods and statements as atomic as a single assignment. During the warm-up phase the generated JHM payload wrapper method gets in-lined and therefore, the microbenchmark loop do actually execute statements. Another important distinction if that AutoJMH uses data extracted from an expected usage of the code, (i.e. the unit tests). Pradel uses randomly generated synthetic data, which may produce unrealistic performance cases. For example, JIT in-lining is a very common optimization that improves performance in the usual case, while reducing it in less usual cases. The performance improvement of this well known optimization is hard to detect assuming that all inputs have the same probability of occurrence.

Program Slicing AutoJMH creates a compilable slice of a program which can be executed, stays in stable state and is not affected unwanted optimizations. Program slicing is a well established field [Tip94]. However, to the best of our knowledge, no other tool creates compilable slices with the specific purpose of microbenchmarking.

4.5 Conclusion and Future Work

In this chapter, we propose a combination of static and dynamic analysis, along with code generation to automatically build JMH microbenchmarks. We present a set of code generation strategies to prevent runtime optimizations on the payload, and instrumentation to record relevant input values for the SUA. The main goal of this work is to support Java developers who want to develop microbenchmarks. Our experiments show that AutoJMH does generate microbenchmarks as accurate as those handwritten by performance engineers and better than the ones built by professional Java developers without experience in performance assessment. We also show that AutoJMH is able to analyze and extract thousands of loops present mature Java applications in order to generate correct microbenchmarks.

Even when have addressed the most common pitfalls found in the current microbenchmarks today, we are far from being able to handle all possible optimizations and situations detrimental for microbenchmark design, therefore, our future work will consist in further improve AutoJMH to address these situations.

Chapter 5

Lossy Compression of ARM Instructions

Code compression techniques [BFG$^+$03] have a positive effect in important economic areas such as embedded systems and Internet of Things/Wireless Sensor Networks (IoT/WSN). Smaller code size reduces memory requirements of embedded systems. In turn, this translates into considerable savings in energy consumption and manufacturing costs [LW06, LHW00, LDK99]. Code compression also enables a reduction in network traffic [ACDFP09], which accounts for an enormous amount of energy consumption in IoT/WSN devices. Therefore, when remotely reprogramming IoT/WSN devices, an efficient code compression scheme for code is considered beneficial or even required.

To the best of our knowledge, all existing code compression schemes are lossless. This somehow limits the ability of a compressor to reduce data size, as the requirement to maintain all information is enforced. The reason for code compression algorithms to be lossless is the traditional belief that code cannot be tampered with without unforeseen consequences. This intuition holds for the most part, however, as discussed in the rest of this thesis, there exist forgiving zones where some instructions can be exchanged by others, still allowing the program to execute and maintain its Quality of Service (QoS) within acceptable levels. The existence of such zones allows the encoder to abandon the lossless requirement when compressing code. Analogous to image compressors that exchange pixels' colors to avoid storing the initial pixel's value, code compressors can replace instructions by similar ones to avoid storing the original.

This chapter introduces Primer, the first (to the best of our knowledge) lossy compression algorithm for ARM assembler instructions. Primer is based on two observations. The first observation is that if some bits in a program are removed, they can be inferred back, knowing how a program should behave. Using this knowledge it is possible to design a function to determine the most likely missing bits' values. We call such function Heuristic Ranking. The second observation is the existence of forgiving zones. If instructed by the user, Primer will exploit forgiving zones to compress programs in a lossy manner: the decompressed program might contain some degree of degradation in the form of bit flips on its assembler instructions and still complete its execution correctly.

Primer is designed as a first step in conjunction with another compressing method (hence its name, referring to primer paint, which improves the lifetime of color coats if applied previously). This is because Heuristic Ranking is able of high compression ratios (up to 1:10), but only for a portion of the program. The rest of the program must be compressed using a third-party algorithm as it is used by the Heuristic Ranking as context to recover the missing bits' values. We found that when using `GZip` to compress data preprocessed with Primer, the compression ratios improved considerably compared with those obtained in unprepared data.

We evaluated Primer using `GZip` as third party algorithm. Our experiments consisted in compressing a set of ARMv5 32-bit programs using `GZip` as baseline and compare the compression ratios with those obtained using both Primer + `GZip` together. Our results show that Premier improved up to 10% the compression ratios obtained using `GZip` alone, while ensuring the decompressed program to execute normally, within acceptable QoS boundaries.

We choose to implement Primer initially for ARMv5 32-bit instructions as ARM is a very popular platform for embedded systems. However, we are confident that the ideas on which Primer is based (Heuristic Ranking and the exploitation of forgiving zones) are also applicable in other architectures as well.

Briefly, the contributions of this chapter are:

- A heuristic to retrieve missing bits in binary code
- A lossy approach that exploits this heuristic to approximately compress bit sequences
- An implementation for ARM
- An evaluation that demonstrates the savings in compression on real world programs

The rest of this chapter explains Primer in detail. Initially, section 5.1 provides some background needed to understand the algorithm's inner working. Primer is then explained in detail in section 5.2. Later on, we provide our evaluation's results in Section 5.3. The chapter provides the related work in 5.4 and concludes in 5.5.

5.1 Background

This section provides the notions and concepts needed to understand the design of Primer. Initially, a brief description of the ARMv5 32-bit Instruction Set Architecture (ISA) is given. We provide only the notions useful to explain Primer's inner working, the complete reference for the ARM assembler language can be found elsewhere [Var17]. Later on, the section introduces two novel concepts specifically designed for Primer: candidate instructions and solution programs. These are key in Primer's design. Both Heuristic Ranking and the Lossy Compression algorithm uses the candidate instructions and solution programs concepts frequently, hence we seek to introduce them early.

5.1.1 ARM ISA

Each ARMv5 32-bits instruction is encoded in a 32-bit integer as shown in Figure 5.1.1 ARM instructions contain one opcode field, one conditional field and one operand field. The opcode field defines what the instruction does. The conditional field defines if the instruction can be executed, which will depend on the value held by the Current Processor Status Register(CPSR). The exception to this rule is the `al` conditional, that makes the instruction execute always. Operands can be one of the following (i) a list of registers, (ii) a list of constant values or (iii) a list of registers and constant values.

Figure 5.1: Simplified view of an ARM instruction

`opcode_{conditional} {operands}`

Examples of valid instructions are:

- `b_eq 0x1234`: Branches (jumps) to address `0x1234` if bit 'equals' (`eq`) is set in the CPRS.
- `str_ne r13 {r4, r5}`: Stores the values of register `r4` and `r5` into the memory pointed by register `r13` if bit 'not equals' (`ne`) is set in the CPRS.
- `mov_al r0, r1`: Moves always the content of register `r1` to `r0`. As the conditional field is '`al`', this instruction executes always, independently of the value stored on the CPRS.
- `subs_al r0, r0, #1`: Subtract always one from `r0` and update the CPSR bits depending on the result of the operation.

To illustrate this within the context of an example, listings 5.1 and 5.2 show, respectively, a simple C++ `for` loop and its translation to ARMv5 32-bit. (The example is taken from [Jac14])

```
for (i = 10; i != 0; i--) {
    myMethod();
}
```

Listing 5.1: A simple C++ loop

```
[0x04]     mov_al   r4, #10
       loop_head_address:
[0x08]     bl_al    myMethod_address
[0x0C]     subs_al  r4, r4, #1
[0x0F]     b_ne     loop_head_address
```

Listing 5.2: The ARM code for the loop in the previous example. The numbers to the left between brackets are the instructions' addresses

The code in listing 5.1 requires little explanation, it is simply a `for` loop that calls the method `myMethod` ten times. The code in Listing 5.2 starts by moving the value 10 into the `r4` register. Then, it branches into the address of the `myMethod` method. After the method returns, the instruction at `[0x0C]` subtracts one from `r4` and updates the CPSR register. Finally, `bne loop_head_address` jumps into the loop head if the 'not equal' bit is set, otherwise it is a NOP instruction and the loop terminates.

5.1.2 Candidate Instructions & Solutions

As mentioned earlier, Primer is based on the observation that it is possible to infer the values of missing bits in the context of a program. Later on, we will see how this is exploited to compress a program. For now, let us take listing 5.2 and assume that some bits' value has been removed from the code. After bit removal, say we obtain the (broken) code of Listing 5.3. The '`??`' characters symbolize the removed parts of the instructions.

```
[0x04]     mov_al    r4, #10
loop_head_address:
[0x08]     bl_al     ??
[0x0C]     subs_al   r4, ??, #1
[0x0F]     b_??      loop_head_address
```

Listing 5.3: The code on listing 5.2 for the loop in the previous example with some bits removed.

Candidate Instruction For each address containing instructions that have had some bits removed (in the example: `[0x08]`, `[0x0C]` and `[0x0F]`) we now have 2^k possible instructions, being k the removed bit count. We call such possible instructions candidates. As this list of instructions contains all possible bit value combinations, it will also contain one special candidate, which is exactly equal to the sequence of bits that has been removed. We call it the original candidate.

For example, suppose that bit 13 at address `[0x0C]` has been removed. As result of this operation, the value of bit 13 is unknown (it can be 1 or 0), therefore, we now have two possible instructions (or candidates) to fill that particular address. The ARM encoding indicates that depending on the value of bit 13, the two possible candidates are `subs_al r4,r4,#1` (if the bit is 0) or `subs_al r4,r6,#1`, (if the bit is 1). The original candidate of address `[0x0C]` is `subs_al r4,r4,#1`, as it is the instruction at that particular address in the original program. In the next section, we will describe how Primer is able to discover this using Heuristic Ranking. Notice that no bits are lost at address `[0x04]`, therefore no candidates spawned here. To generalize this special case, we say that address `[0x04]` contains only one candidate: the original candidate.

Solution Program A solution program is one of the many possible programs that can be formed selecting a single candidate per address. In other words, a solution program is a program built by selecting one candidate per address.

5.2 Approach

In this section we explain Primer in details. We begin by illustrating the principle on which Primer is based to compress code using Heuristic Ranking. Then we explain the algorithm, both compression and decompression schemes. Finally, we go into details explaining Primer's components in the following order: (i) Heuristic Ranking function in 5.2.3 (ii) the Successive Packing Algorithm (SPA) [SZ02] in 5.2.4 (iii) lossless encoding in 5.2.5 and (iv) we conclude this section explaining the lossy encoding in 5.2.6.

Table 5.1: Using Heuristic Ranking to compress code. The table shows all the candidates spawning after the removal of 6 bits, the same candidates already sorted by rank and the rank assigned to each candidate by Heuristic Ranking. Finally, the table also shows the number of bits required to store the rank.

Candidates spawn after bit removal	**Candidates Sorted by rank**	Rank	Size (bits)
stmdb sp, {r3,lr}	**push {r3,r8,lr} ← Lossy!**	0	0
stmdb sp, {r3,r8,lr}	**push {r3,lr} ←Original**	1	1
stmdb sp, {r3,r9,lr}	push {r3,r9,lr}	2	2
stmdb sp, {r3,r8,r9,lr}	push {r3,r8,r9,lr}	3	2
ldmdb sp, {r3,lr}	stmdb sp, {r3,lr}	4	3
ldmdb sp, {r3,r8,lr}	stmdb sp, {r3,r8,lr}	5	3
⋮	⋮	⋮	⋮
push {r3,lr} ← Original	ldmdb sp, {r3,lr}	15	4
push {r3,r8,lr}	ldmdb sp, {r3,r8,lr}	16	4
⋮	⋮	⋮	⋮
push {r3,r8,r9,lr}	ldmdb sp, {r3,r8,r9,lr}	30	5
ldmdb sp!, {r3,lr}	ldmdb sp!, {r3,lr}	31	5
⋮	⋮	⋮	⋮
ldmdb sp!, {r3,r9,lr}	ldmdb sp!, {r3,r9,lr}	127	6
ldmdb sp!, {r3,r8,r9,lr}	ldmdb sp!, {r3,r8,r9,lr}	128	6

5.2.1 Compression's Principle

Primer compresses a program removing bits from it. As result of this bit removal, we now have a series of candidates for each address. To recover the original candidate, Primer uses Heuristic Ranking to rank (sort) all possible candidates by their likelihood of being the original candidate. The compression in Primer comes precisely from storing this rank, instead of the removed bits. To better understand this, take the real example of Table 5.1, obtained from the `bitcount` program in our dataset. Primer has removed six bits from instruction `push {r3,lr}`, yielding $2^6 = 128$ candidates. Afterward, the Heuristic Ranking is used to sort all candidates. In the example the rank assigned to the original candidate is 1, a number requiring a single bit to be stored instead of the original 6 bits.

Lossy Compression Principle The optimal result for Heuristic Ranking is to give rank zero to the original candidate, effectively requiring no information to retrieve the removed bits' values. However, our current implementation is not always able to give rank zero to the original candidate. This is where the Lossy Phase starts. The Lossy Phase tries to replace all original candidates in the program by candidates that are better ranked. The intuition here is that smaller ranks require fewer bits to be represented. If the program executes successfully using a better ranked instruction, the Lossy Phase uses the better rank instead of the original, reducing the number of bits required to store the program. In the example, if the program still executes normally using instruction `push {r3,r8,lr}` instead of `push {r3, lr}` then no information is needed to represent the removed bits.

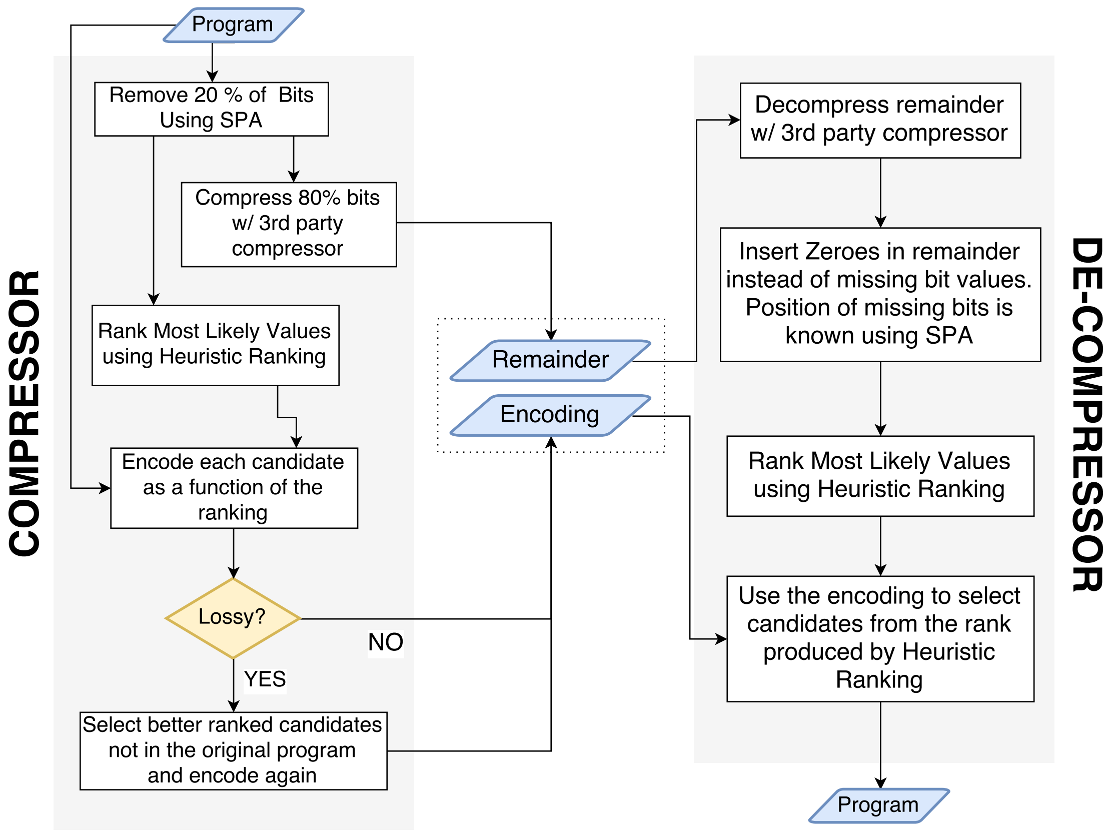

Program
Remove 20 % of Bits Using SPA
Compress 80% bits w/ 3rd party compressor
Rank Most Likely Values using Heuristic Ranking
Encode each candidate as a function of the ranking
Lossy?
NO
YES
Select better ranked candidates not in the original program and encode again
COMPRESSOR
Remainder
Encoding
Decompress remainder w/ 3rd party compressor
Insert Zeroes in remainder instead of missing bit values. Position of missing bits is known using SPA
Rank Most Likely Values using Heuristic Ranking
Use the encoding to select candidates from the rank produced by Heuristic Ranking
Program
DE-COMPRESSOR

Figure 5.2: Overall view of the compression and decompression schemes of Primer. In the image's left side is the compression scheme. Its output is passed to the decompressor, which reverts the process and produces a working program.

5.2.2 Algorithm

Figure 5.2 shows the compression and decompression scheme for Primer. As input, the algorithm receives a program P and a user-defined parameter to set lossy compression on/off. The compression scheme's output is a bit stream requiring less storage space than the original code. In turn, the decompressor's consumes this bit stream and produces a working program.

For compression, Primer creates a copy P' of P. Then it removes up to 20% of the bits in P'. Removed bits are selected using SPA, a strategy that preserves context to increase Heuristic Ranking's efficiency. Afterwards, the algorithm employs Heuristic

Ranking to sort all candidate instructions in each address of P' by their likelihood of being the original instruction for that same address in P. In a later stage, the Lossless Encoder uses the original program P to determine the index of the original candidates in the sorted list of candidates produced by Heuristic Ranking. These indexes (or ranks) are then used to encode the solution. Afterwards, if the algorithm is configured to be lossy, the Lossy Phase starts, further reducing the encoding size by selecting better ranked candidate instructions not in the original program. In a final step the remaining 80% of P' is compressed using a third party algorithm and stored in a bit stream we call remainder. The output of the compression scheme is the encoding and the remainder joined together.

The decompression scheme starts by decompressing the remainder using the third-party algorithm. This results in an uncompressed bit stream which is still considerably smaller than the original program, as the bits removed by Primer are still missing. In a second step, zeros are inserted in the position of each removed bits. This 'inflates' the code to its original size. At this stage, the program is still not functional, as all inserted bits values are zero. In a final step, the Heuristic Ranking Function is invoked again, all possible solutions are sorted in the decompressor and the encoding is used to determine which solution is equal to the original program.

5.2.3 Heuristic Ranking Function

Heuristic Ranking is a fundamental part of Primer. The whole algorithm is designed around this function. Hence, we introduce this concept first and then we go into details of all the remaining components.

The Heuristic Ranking function ranks (sorts) each candidate instructions by the candidate's probability of being the original one. This is done using probabilistic inference with a carefully chosen set of events to determine a candidate's probability of being the good one, as shown in Equation 5.1 (being $I_{k\alpha}$ the k-th candidate at a given address α.):

$$ranking(I_{k\alpha}) = P_B + P_{SL} + P_T \tag{5.1}$$

Where:

$$P_B = P(B|\Phi, J) \qquad P_{SL} = P(SL|\Phi, RU, SLP)$$
$$P_T = P(SL', B'|\Phi, RU, RD) \qquad \Phi = Cn, Op, Cf$$

Equation 5.1 is in fact a Bayes Net that scores the candidates based on factors such as: the expected distribution on a program of Conditional fields (event Cn), Opcode fields (event Op) and Registers (event RU). It also considers if placing a candidate in a given address will create weird or exceptional control flow (event Cf) or data flow (event RD). Finally, memory access (event SLP) and jumping addresses (event J) are observed as well.

The equation discerns between branch instructions (P_B), Store/Load instructions (P_{SL}) and anything that is neither branch nor Store/Load (P_T). This is done to exploit the knowledge we have on the semantics of each individual classes. We model events B, SL and SL' to be mutually exclusive, therefore $P_B + P_{SL} + P_T \leq 1$.

The probabilities of all these events are known by observation in a set of uncompressed programs [1]. These values are embedded in Primer and are not transmitted with the compressed file, except when they diverge from the embedded model by a value greater than $1/2^{k\alpha}$, being $k\alpha$ the candidate count of the address with most candidates. Notice that $1/2^{k\alpha}$ is the probability of any candidate of being the good one in absence of any other information in address α.

The rest of the section is devoted to expand on these events and the rationale behind them.

Events *Cn*, *Op* and *RU* (Instruction Distribution) Cn, Op and RU are (respectively) the events of placing a candidate $I_{k\alpha}$ at address α and having the resulting solution program maintain the distribution of conditionals, opcodes and registers initially observed in the uncompressed programs.

Intuition: The distribution of conditionals, opcodes and used registers is observed in the uncompressed programs, any candidate making the solution fit such initially observed distribution has a fair amount of chance of being the good candidate and is rewarded accordingly with a high score.

Event *Cf* (Good Control Flow) Cf is the event of placing candidate $I_{k\alpha}$ at address α and having the formed solution program hold the following conditions: (i) instructions with same conditionals fields are grouped together (ii) a conditional branch occurs only shortly after modifying the CPRS register.

Intuition: Programs are constructed by execution blocks. The ARM ISA has two ways of constructing an execution block: (i) by grouping instructions with same conditionals together or (ii) by having a conditional branch after modifying the CPRS register. The Heuristic Ranking awards instruction causing the program abide to this expected behavior with a high score.

Event *RD* (Good Data Flow) RD is the event of placing candidate $I_{k\alpha}$ at address α and having the solution program hold the following: (i) registers are only read after being written (ii) written registers are read afterward.

Intuition: Reading from registers that have been not written might result in unwanted results. Also writing to registers that are not read afterwards is unusual, as registers are scarce resources. Therefore, the Heuristic Ranking rewards a high score to candidates that reduce the solution programs' chance of exhibiting these unexpected behaviors.

Event *J* (Good Jumping Addresses) J is the event of selecting a candidate $I_{k\alpha}$ at address α and have a resulting solution program hold the following: (i) $I_{k\alpha}$ is a branch (ii) the jump address points to the beginning of a function or to an address within the

[1] All programs in this dataset are described in Section 4.3

function containing the instruction. (Notice that the beginning and length of a function are approximated as they are obtained from the ELF symbol table).

Intuition: Branches are used for function calls or to create control flow structures within a function. Therefore, Heuristic Ranking will favor branches jumping the beginning of a different function (calls) or to the same function (creating control flow). The Heuristic Ranking will penalize branching to the middle of a different function or outside the program.

Event *SLP* (Store Register Pair) *SLP* is the event of placing a candidate $I_{k\alpha}$ at address α and having (i) $I_{k\alpha}$ is a store/load instruction (ii) if the $I_{k\alpha}$ is a store, it writes the registers' content to memory at the beginning of a function OR if the $I_{k\alpha}$ is a load, it reads registers values from memory (iii) if the $I_{k\alpha}$ is a store writing registers to memory at the beginning of a function, there is a corresponding load reading them from memory at the end OR vice-versa, if the $I_{k\alpha}$ is a load reading registers from memory, there is a corresponding store writing to memory at the beginning of such function.

Intuition: Depending on the calling convention, the caller or the callee must preserve the status of registers. In any case, this will result in store/load pairs preserving/recovering the value of registers near a function call. Because of this, the Heuristic Ranking will reward these store/load pairs with a good score.

5.2.4 Bit Removal

To increase Heuristic Ranking efficacy, bits must be removed scattered around the code. Heuristic Ranking uses the context surrounding candidates to score them, hence Primer does its best effort to preserve such context. This is done using a carefully designed bit removal strategy. To exemplify the previous intuition, we apply three different removal strategies in the example of listing 5.2.

```
[0x04]    ????????????
[0x08]    bl_al   myMethod_add
[0x0C]    subs_al r4, r4, #1
[0x0F]    b_ne    loop_head_address //0x08
```

Listing 5.4: Bad Strategy 1: Remove bits as a consecutive stream. Preserves no context

Bad strategy of Listing 5.4 removes bits consecutively. As result of this, no information regarding the removed instruction is preserved, making almost impossible to recover it.

```
[0x04]    mov_ ??   r4, #10
[0x08]    bl_  ??   myMethod_add
[0x0C]    subs_??   r4, r4, #1
[0x0F]    b_   ??   loop_head_address //0x08
```

Listing 5.5: Bad Strategy 2: Remove bits at a fixed position. Preserves little context

Bad strategy in Listing 5.5 removes bits always at the same position. In here, all information regarding the program's control flow is destroyed, disabling recovery based on the expected program's control flow behavior.

```
[0x04]     ??_al    r4, #10
[0x08]     bl_??    myMethod_addrs
[0x0C]     subs_al r4, ??, #1
[0x0F]     b_ne     ??_head_addrs //0x08? 0x8^32?
```

Listing 5.6: Good Strategy: Removes bits scattered around the code. Preserves the most context

This result is better, as it preserves more context. In here, the removed opcode at address `[0x04]` can be recovered using the opcode fields expected distribution; the removed ‘`al`’ conditional at address `[0x08]` can be recovered using the intuition of event Cf: instructions with same conditionals are grouped together; the register lost at `[0x0C]` can be recovered knowing that it must have been written earlier (intuition of event RD) and therefore it must be `r4`. The higher bits of the jumping address at `0x0F` can be recovered by discarding jumping addresses outside the program.

The previous examples motivate the need for a number generator able to generate uniformly scattered positions, having two consecutive generated positions as distant as possible from each other. We describe such generator as follows.

The Successive Packing Algorithm (SPA) The Successive Packing Algorithm (SPA) [SZ02] method was originally designed to interleave 2D images and mitigate burst errors during network transmissions. The SPA also deals with the problem of mapping elements of 2D structure into a 1D series (in their cases packets, in ours, the removed bit list) in such a way that nearby elements in the 2D structure (in their case pixels in an image, in our case bits in an instruction list) are as distant as possible in the 1D series.

The original algorithm starts with a 1×1 matrix $S^0 = [s_0]$, which is then successively used as input for the next matrix, generated as follows:

$$S^{i+1} = \begin{bmatrix} 4 \times S^i + 0 & 4 \times S^i + 2 \\ 4 \times S^i + 3 & 4 \times S^i + 1 \end{bmatrix}$$

According to the algorithm $S^0 = [s_0]$, while S^1 and S^2 are:

$$S^1 = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix} \qquad S^2 = \begin{bmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{bmatrix}$$

We use SPA to create a 32×32 matrix S^5, where cell S^5_{ij} represents the i-th bit of the j-th instruction. Obviously, this matrix will cover only the first 32 instructions of the program. If the program is bigger than that, more matrices are created using SPA until the whole program is covered, concatenating the newly built matrices' rows to form a $C \times 32$ matrix, (where C is the address count). In a final step, the first $C/2$ even rows are interchanged, so that row i is exchanged with row $C/2 + i$. This is done to further increase the distance between two consecutive positions. The matrix for the example of listing 5.2 is:

$$S^5 = \begin{bmatrix} 0 & 512 & 128 & \dots & 554 & 170 & 682 \\ 960 & 448 & 832 & \dots & 490 & 874 & 362 \\ 192 & 704 & 64 & \dots & 746 & 106 & 618 \\ 768 & 256 & 896 & \dots & 298 & 938 & 426 \end{bmatrix}$$

Generating the Sequence of Bits to Remove Armed with the $C \times 32$ matrix Primer is now able to generate the sequence of bits to remove by looking-up in the matrix the positions of the first consecutive k numbers, (where k is equal to the 20% of the total bit count in the program) the column containing the number represents the bit index to remove and the row represents the instruction from which the bit is going to be removed.

Knowing the number of bits to remove As mentioned earlier, the Heuristic Ranking is used to obtain a value to encode removed bits. To effectively compress the program, this encoding should require less storage space than the removed bits. As we remove larger amounts of bits, the lack of contextual information causes the Heuristic Ranking to be less accurate when determining the most likely candidates, increasing the size of the encoding by means of worse ranks awarded to original instructions. In the programs used for our evaluation, we discovered that the best trade-off between the number of bits to remove and the resulting encoding's size was to remove around 20% of the program bits.

5.2.5 Encoding

Recall that a solution program is one of the many possibles programs that can be formed by selecting one candidate per address. Primer encodes the missing bits using Equation 5.2. In this equation, α is an array containing all instructions' addresses of P in consecutive order (i.e. `0x04`, `0x08`, `0x0C`,...) k_i is the rank awarded by the Heuristic Ranking to the original candidate at the i-th slot of α and C is the length of the α array. Equation 5.2 is nothing but a positional numeral system that uses as base B the largest rank awarded to a candidate (just like decimal system uses base 10 and the binary uses base 2) and as digits the ranks awarded to candidates.

$$E = \sum_{i=0}^{C-1} k_i B^i \tag{5.2}$$

Where

$$k_i = HeuristicRank(\alpha[i]) \qquad B = max(k_i) + 1$$

We explain now how the original solution is encoded using this equation. To do so, we use once more the motivating example of listing 5.2. Suppose that Primer removes seven bits from addresses [0x04] and [0x08], nine bits from address [0x0C] and eight bits from address [0x0F]. Notice that 31 bits (the 24.2%) have been removed from the program. Using Heuristic Ranking, Primer sorts all the candidates by their likelihood of being the good one. Say the original candidates gets ranked at positions 1, 0, 2 and 2 at addresses [0x04], [0x08], [0x0C] and [0x0F] respectively. The maximal rank awarded are in addresses [0x0C] and [0x0F] and is equal 2. Hence $B = 2 + 1 = 3$. Substituting in 5.2 we have:

$$E = 1 \times 3^3 + 0 \times 3^2 + 2 \times 3^1 + 2 \times 3^0 = 35$$

The resulting encoding value of the example (35) can be expressed with 6 bits rather than the 31 bits removed from the program.

```
Data: $G$: Dictionary containint the list of candidates per address;
      $S$ Dictionary containing the good candidates indexes per address;
      $PROG$ Program to be modified
      $QoS$ Quality of Service function
Result: A lossy encoding for $PROG$
1  foreach $\alpha \in G$ do
2      foreach $k \in [0..S[\alpha]]$ do
3          $PROG[\alpha] = k$
4          if $QoS(PROG)$ is TRUE then
5              $S[\alpha] = k$
6              break the loop
7          end
8          $PROG[\alpha] = S[\alpha]$
9      end
10 end
```

Algorithm 3: The Primer's Lossy Phase greedy algorithm

5.2.6 Lossy Compression

The lossy compression in Primer is a greedy algorithm that tries to modify the program being compressed to make it use a candidate better ranked by the Heuristic Ranking function in a given address. Smaller ranks require less memory, therefore using better ranked candidates further reduces the encoding's size.

In the previous example, say the Lossy Phase discovers that the program can maintain its QoS using the candidate sorted at position 0 at address [0x04], then the encoding will become equal to:

$$E = 0 \times 3^3 + 0 \times 3^2 + 2 \times 3^1 + 2 \times 3^0 = 7$$

Which reduces to 3 bits the required amount of storage, improving the example's lossless solution 57%. Notice that by replacing a single candidate, the Lossy Phase is able to considerably reduce the amount of information needed to encode the program.

The method used by the Lossy Phase is shown in Algorithm 3. It takes the following parameters: a dictionary G containing the list of all candidates per address, where each list is already sorted using Heuristic Ranking; a dictionary S containing the good candidate index per address (i.e. the good solution); the program $PROG$ to compress and the QoS function. The lossy algorithm goes through all the addresses of the program (line 1), trying to use the best ranked instruction possible. In line 2, the algorithm begins with the best ranked candidate and tries all instructions up to the good one, which is stored in $S[\alpha]$. The way this is done is by overriding the program at address α with the better ranked candidate. If program complies with the QoS requirements, then such better ranked candidate is used, the inner loop breaks and the next address is explored. On the other hand if the modified program's QoS is not acceptable, the algorithm rolls back the last change (line 8 and moves on into the next candidate.

The Premier's lossy compression strategy is greedy. It starts by the most significant digits of the encoding E, which is always the best solution since:

$$xB^n > \sum_{i=1}^{n} y_i B^{n-i} \quad \forall x, y_i < B$$

5.3 Evaluation

Our evaluation has four main objectives. Initially, we seek to evaluate Heuristic Ranking, the main component of Primer and observe how good is the function in terms of giving the lowest possible ranking to original instructions. Later, we want to understand the reasons for which the Lossy Phase of Primer is able to further improve compression ratio without affecting the dataset program's QoS. Afterwards, we determine whether Primer can improve over existing compression schemes Finally, we observe the effect of the lossy phase on the compression ratio of the data. With these objectives in mind, we propose the following Research Questions:

RQ1: Can indeed Heuristic Ranking provide small ranks to original instructions?

The compression principle of Primer is based on the assumption that Heuristic Ranking will assign a small rank value to the original candidate. The algorithms then uses this rank to encode bits removed from the instruction. Smaller numbers require less storage space, therefore assigning lower ranks values to the original candidates reduces the number of bits required to store it. We seek to observe whether Heuristic Ranking can indeed award small ranks to original candidates and also the number of bits required to store each rank.

RQ2: Why does the Lossy Phase of Primer further reduce file size without affecting the program's QoS?

In order to better understand why Primer is able to find equivalent programs, we wanted to characterize the forgiving zones found. We do so to improve future versions of the algorithm, by prioritizing zones abiding to patterns known to be forgiving.

RQ3: Can Primer provide better compression ratios than existing industry-standard algorithms such as `GZip`?

This is the main validation of the Algorithm, as the resulting size of the compressed file is one of the main metrics on which compression algorithms are evaluated. There is no point in introducing a new algorithm that does not improve upon existing results.

RQ4: What is the impact of Primer's lossy phase in the compression ratio?

The Lossy phase of Primer modifies the original sequence of instructions, in a way that loses information. Primer guarantees that the decompressed program executes normally and that the user-defined QoS metric is met, but admittedly this approach conveys risks. Therefore, without an improvement in compression, the lossy phase of Primer lacks in appeal.

Table 5.2: Case studies for our experiments

Program	Code Size in Bytes	Purpose
`sha`	3096	Library implementing the Set of Hash Algorithm (SHA-0)
`qsort_small`	1076	A Program sorting several vectors using the C Standard Library function `qsort`
`fft`	8356	Library with the implementation of the Fast Fourier Transform
`dijkstra_small`	2292	Implementation of the Dijkstra's Algorithm
`crc32`	1140	Implementation of the Cyclic Redundancy Check (CRC) error-detecting codes
`bitcount`	5880	Test program for bit counting functions
`basicmath`	7244	Test program with basic computations

5.3.1 Dataset

We compressed individually seven programs from the MiBench suite [GRE+01]. MiBench is a collection of representative embedded applications and libraries. We selected the suite since the initial intended usage of Primer is precisely over-air firmware updates of embedded systems. Table 5.2 shows the programs of our dataset.

All sources for the programs in our dataset were obtained from the MiBench website [2]. In a second step, these sources were compiled for the ARMv5 32-bit ISA. We used Primer in lossless mode to compress the resulting machine instructions. In a second step, we repeated the compression process using Primer in lossy mode. Finally, we also compressed all machine instructions using `GZip` to obtain a baseline that we could use to compare against Primer.

5.3.2 RQ1: Quality of Heuristic Ranking

The number of bits needed to encode a rank r is equal to $log_2(r)$. Candidates awarded with rank 0 requires zero bits to encode and those awarded with rank 1, require one bit. The highest rank awarded to a candidate in our dataset was 13. In other words, 4 bits was the maximal amount of information needed to encode any candidate in our dataset.

[2]`http://vhosts.eecs.umich.edu/mibench/`

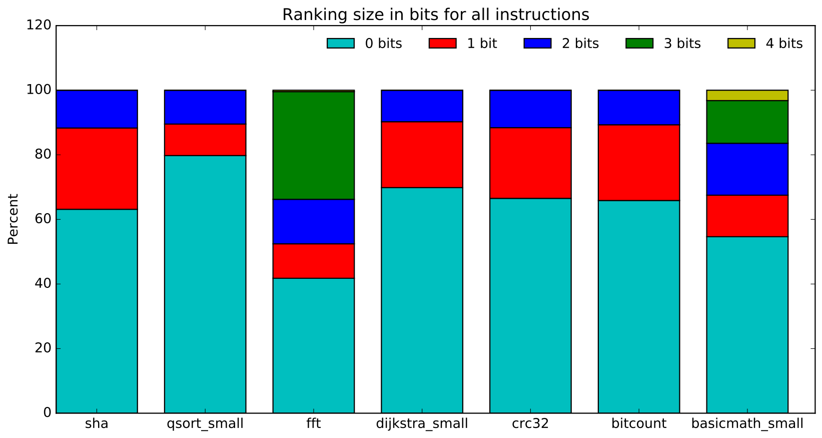


Figure 5.3: Number of bits required to encode original instructions in our dataset. Each segment represents the percent of instructions requiring from 0 to 4 bits to be encoded. In our experiments between 6 and 8 bits were removed from each instruction

In our experiments we removed 20% of the bits in all programs of our dataset. This means that between 6 and 8 bits were removed from each instruction. The number of bits eliminated per instruction varied since the SPA generator does not assigns an even quantity of bits to remove at each address.

Figure 5.3 shows the percent of original instructions that the Heuristic Ranking was able to encode using 0, 1, 2, 3 and 4 bits. The figure shows a bar for each program in our dataset. In turn, each bar is divided into segments whose length indicates the percent of instructions that Heuristic Ranking was able to encode using a determinate number of bits.

The image gives us an idea of the Heuristic Ranking's performance. In all programs, at least 40% of original instructions was awarded rank 0. In program `qsort` nearly the 80% of original instructions was awarded rank 0. Also, only in programs `fft` and `basicmath` a minor percentage of instructions required 4 bits to encode. This clearly indicates the Heuristic Ranking's capacity to determine which is the most likely instruction to live in a given address.

We believe this good performance is mainly due to the effectiveness of the probabilistic model on which Heuristic Ranking is based. Other factors that we have observed are the fact that many candidates were not valid ARM instructions and others were simple to discard, such as branches to invalid addresses, storages without matching loads and useless computations stored in registers that were never used afterward.

Answer to RQ1: The Heuristic Ranking is able to effectively award near-zero ranks to original candidates.

5.3.3 RQ2: Characterization of Forgiving Zones exploited by Primer

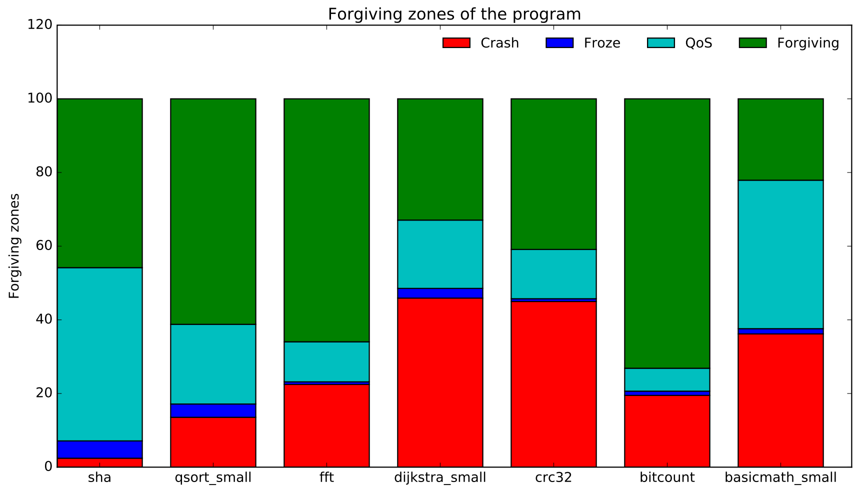


Figure 5.4: Size of forgiving zones all programs in our dataset

Recall that Primer achieves compression by storing the rank assigned to the original candidate by the Heuristic Ranking function, instead of the removed bits. Smaller ranks require less memory, therefore Primer's Lossy Phase tries to store the bests ranked candidates, independently of whether that candidate is equal to the original instruction in the program being compressed. By storing a smaller rank, higher compression levels are achieved. Obviously, this is only possible if the candidate being used allows the program to execute normally while maintaining its QoS within acceptable bounds.

To better understand why Primer's Lossy Phase is able to further improve compression ratio, we now give a closer look to the forgiving zones exploited by Primer. Initially we provide some insights on the size of such zones. Later on, we show some examples of instructions exchanged by Primer's Lossy Phase and comment on the reasons why substitution works.

Size of Forgiving Zones Figure 5.4 shows which parts of the programs in our dataset were forgiving. The figure shows one stacked bar per program. Each bar is divided in four segments 'Forgiving', 'QoS', 'Crash' and 'Froze'. In turn, each segment represents a percentage of the program's instructions that were forgiving, crashed or froze the program or that produced a different output.

Instructions in the Forgiving Zones were those instructions that could be exchanged by at least another candidate. The non-forgiving instructions were divided in three groups 'Crash', 'Froze' and 'QoS'. Crash instructions were those that crashed the program in most attempts of being exchanged. Similarly, Froze instructions where those for which a replacement usually prevented the program from exiting at all. Finally, QoS instructions were those that produced a different output in most attempts of replacement. If an instruction caused more than one type of error, we selected the most common one. For example, if an instruction was replaced five times and three replacements caused the program to crash, we placed the instruction in the Crash group.

Figure 5.4 shows that a large portion of the program could be exchanged by at least another candidate. In four cases (`sha`, `qsort`, `fft` and `bitcount`) the forgiving zones accounted for more than 50% of the instructions. The programs with larger forgiving zones are `fft` and `bitcount`. Not surprisingly, those were also the programs for which the Lossy Phase of Primer improved the compression ratio the most.

Examples of Forgiving Zones Figure shows 5.4 that considerable portions of our dataset's programs contain instructions that can be exchanged without affecting the program's behavior. Following, we provide some examples of the replacement patterns that we have observed Primer to perform in the Forgiving Zones.

- Unused or exchangeable conditionals. We have found that some instructions are conditioned to situations that never occur, like overflow in operations with bounded values. It can also happen that in a particular context two conditionals are exchangeable such as 'minus/negative' (`mi`) and 'signed less than' (`lt`), which are the same if the overflow bit is clear in the CPSR register. Listing 5.7 shows an example where the conditional of instruction `eors_gt` and parts of the instruction `strbt_vs` can be exchanged without consequences, as no overflow occurs.

```
// ORIGINAL PROGRAM:
[0x10d88] eors_mi r0, r6, r0
[0x10d8c] eors_gt r0, pc, r0
// The following executes only if overflow:
[0x10d90] strbt_vs r6, [r6], -r6, ror #12

// MODIFIED PROGRAM:
[0x10d88] eors_lt  r0, r6, r0 //<-CONDITIONAL EXCHANGED(mi by lt)
[0x10d8c] eors_gt  r0, pc, r0
[0x10d90] strbt_vs r5, [r6], -r6, ror #12 //<-REGISTERS CHANGED
```

Listing 5.7: The original program is on top, while the modified program is at the bottom.

```
// ORIGINAL PROGRAM:
[0x10c88]  str r3, [fp, #-0x34]  // <-- MODIFIED JUMPS HERE
[0x10c8c]  str r4, [fp, #-0x30]
[0x10c90]  mov r2, #0            // <-- ORIGINAL JUMPS HERE
[0x10c94]  ldr r3, [pc, #0x148]
[0x10c98]  sub r1, fp, #0x34
```

Listing 5.8: Example of a case where two extra instructions were executed by the modified program without consequences. The original branch transfer execution to `[0x10c90]` while the modified program transfered execution to `[0x10c88]`.

- Small variations in branching addresses. If small variations are introduced in the branching addresses, causing the program to jump into one or two instructions before, there is a chance that the extra executed instructions have no impact in the program. Listing 5.8 shows a part of the program where one of such modified branch instructions branched to. The original instruction branches to address [0x10c90] while the modified one branched to [0x10c88]. The two extra executed instructions stored the value of registers r3 and r4 to memory without major consequences to the program.

- Store unnecessary register values. Depending on the calling convention the caller or callee must preserve register values at the beginning of a method and restore such values at the end of a method's execution. Sometimes however, preserved values are unused after the method returns. Therefore, a common substitution pattern is to replace registers in the list of preserved registers in push or pop instructions. Listing 5.9 shows one of such exchanges found in basicmath, one of our dataset's programs.

```
// ORIGINAL PROGRAM:          // MODIFIED PROGRAM:
[0x105d0]   push {r4, lr}     [0x105d0]   push {r1, r4, lr}
[0x105d4]   blx  r3           [0x105d4]   blx  r3
```

Listing 5.9: To the left is the original program, to the right the modified version. The modified program stores one extra register (r1) to memory.

- Replacement of instructions by equivalent ones. One common pattern found in Primer's replacements was instructions whose execution will yield equal results. Listing 5.10 shows one of those substitutions. In the original program, a constant value zero is moved into r2, while the modified program subtracted r2 from itself and store the value in r2, effectively storing the value zero in r2.

```
// ORIGINAL PROGRAM:                // MODIFIED PROGRAM:
[0x10c6c] mov r2, #0                [0x10c6c] sub r2, r2, r2
[0x10c70] ldr r3, [pc, #0xf4]       [0x10c70] ldr r3, [pc, #0xf4]
[0x10c74] sub r1, fp, #0x34         [0x10c74] sub r1, fp, #0x34
```

Listing 5.10: To the left is the original program, to the right the modified version. Primer exchaged an instruction by one yielding the same result.

Answer to RQ2: Primer's Lossy Phase is able to reduce compressed program's size exploiting large forgiving zones found in each program. It also profits from frequent opportunities to exchange instructions without affecting a program's QoS.

5.3.4 RQ3: Comparing Primer with Industry Standard GZip

Figure 5.5 shows the results of our experiments comparing GZip with Primer. In the figure, each program in our dataset is represented by three bars that indicate (from left to right) the inverted compression ratio (i.e. *compressed_size/uncompressed_size*)

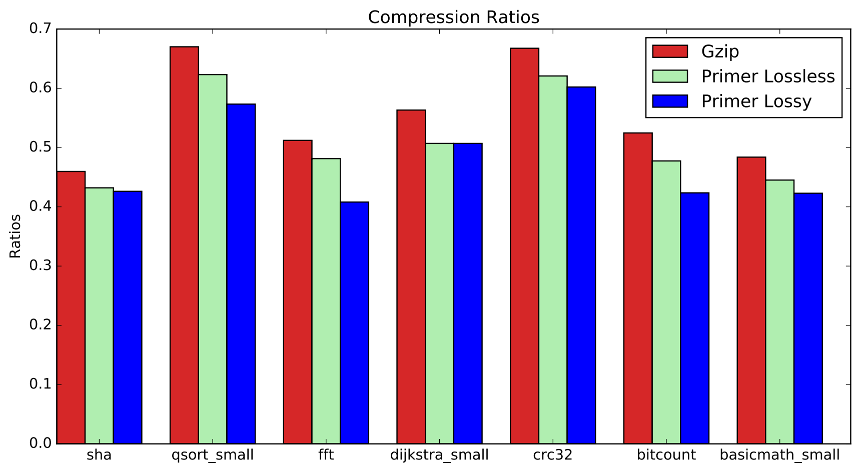


Figure 5.5: Compression ratios of Gzip, Lossless Primer and Lossy Primer

obtained using `GZip`, Primer in lossless mode and Primer in lossy mode, respectively. As the bar's height represents the inverted compression ratio, lower is better.

In the experiments, we initially compressed each program with using `GZip` alone. The resulting files were used as baseline. Later on, we preprocessed up to the 20% of each program using Primer Lossless and then Primer Lossy, the remaining 80% of each preprocessed program was compressed using `GZip` and appended to Primer's output to obtain a compressed file containing the complete program. The size of this compressed file was then compared with the baseline. The results are shown in Figure 5.5.

The results demonstrate that Primer was able to consistently produce smaller outputs. The algorithm was always able to improve the compression ratio obtained using `GZip` alone. This is due to the fact that Primer removes a part of the file and compresses it using a technique which is better aimed at compressed code than the general purpose scheme used by `GZip`. Notice also the different improvement ratios obtained using Primer. These differences are mainly caused by how much a program abides to Heuristic Ranking's probabilistic model. This is somehow expected, the more a program deviates from the model, the more information is required indicate where and how much it deviates from such model.

Answer to RQ3: Primer is able to improve the compression ratios obtained with the industry-standard tool `GZip`.

5.3.5 RQ4: Lossyness' Impact on Compression

Primer's Lossy Phase tries to modify the program being compressed to make it use candidates having a better rank than the original candidate. This is done by replacing candidates instructions and then checking if the modified program still meet the QoS metric. In our experiments we used a simple QoS metric that consisted in verifying if the approximated program produced an output equal to the original. Only if both outputs matched, the QoS was satisfied.

Figure 5.5 shows that the Lossy Phase was able to improve the compression ratio in six out of seven programs in our dataset. The improvements obtained over the Lossless phase ranged between 2% and 10%. Programs `fft` and `bitcount` received the highest improvements from the lossy phase. On the other side, programs `sha` and `crc32` received lesser improvements. Finally, program `dijkstra` received no improvement at all from the Lossy Phase.

The improvements on compression ratio yielded by the Lossy Phase of Primer depended mainly on the phase's ability to find an instruction lying in a forgiving zone of the program. Forgiving zones' sizes differ from program to program (as shown in Section 5.3.3). Not surprisingly, the Lossy Phase yielded larger compression ratio improvements precisely on those programs having the largest forgiving zones.

Another important factor influencing the reduction in size achieved by the Lossy Phase was the encoding Equation 5.2. Recall that the solution's encoding consists in a positional numeral system, where the ranks of the candidates are used as digits. This means that, like in numeral systems where there are high order and low order digits, there will be 'high order' and 'low order' instructions, having high order instructions larger impact in the encoding's size. In the example of Section 5.2.6 we saw how the Lossy Phase made a significant impact in the compression ratio replacing a single high order instruction. We observed a similar behavior in our experiments. By targeting always the instruction in the most significant positions of the encoding, the Lossy Phase was able to significantly reduce encoding size replacing only a few candidates.

Answer to RQ4: The Primer' Lossy Phase was able to further improve the compression ratio of six programs (out of seven) in our dataset.

5.3.6 Threats to validity

While the results obtained using Primer show promise, there are some threats to the validity of the findings shown in this chapter. Firstly, the Heuristic Ranking uses a probabilistic model that can potentially be overfitted to the programs in the dataset. We hope this is not the case, as the programs used to evaluate the algorithm belong to a representative benchmark specifically designed for embedded systems. The second threat to validity is that our results were obtained using a complex tool chain. We not only implemented Primer, but also tools to disassemble programs and analyze data such as the size of the forgiving zones. While we tested our tool chain, extensively, we do not discard the existence of bugs. Hopefully, if any, such errors will not distort the main character of our findings.

5.4 Related Work

To the best of our knowledge Primer is the first lossy technique specifically designed to compress code. The algorithm is primarily meant to reduce network traffic in IoT/WSN devices [ACDFP09] (specifically when updating software remotely). Therefore, we review other techniques addressing this particular problem [BS13]. More generally, code-size reduction is a well established field of studies [BFG$^+$03] with many related works in the area. Closer to Primer are the work by Frazer[Fra99], also Code Compaction schemes and compression algorithms for embedded systems.

Reducing network traffic in IoT/WSN Data compression is only one of the many strategies used in IoT/WSN to reduce network traffic. Other approaches include data prediction, sampling or sending data only when the best conditions exist. Techniques for data prediction tries to avoid traffic by sending only data that does not fit to a particular predicted behavior [CDHH06, GPI06]. Sampling techniques collect and sends only a subset of the data based on certain criteria [GLY07, AAFR10]. Other authors propose to wait for optimal conditions to send data, such as the sender and receiver node having nearby locations on mobile networks [WBMP05, JZA$^+$05].

Reprogramming IoT/WSN Many techniques for reprogramming IoT/WSN devices give special attention to errors produced during update transmission [**?**, SV06]. Such errors can potentially increase traffic since corrupted packets need to be resent. Therefore, authors in this field propose techniques able to recover from errors and thus reduce traffic. Other approaches take into consideration that an update is most likely incremental (i.e. the old and new version of the program are likely to share large portions of code) and exploit this to send only diverging parts and reduce traffic [JC04, RW08]. This incremental update is easy to implement if the program is highly modular [LC02].

Code Compression Algorithms for Embedded Systems A significant number of papers have proposed code compression techniques specifically meant for embedded systems [BMMP99, LHW00, ZK06]. The objective here is to reduce memory requirements [BMMP99], which in turns brings down costs [ZK06] and energy consumption [LHW00]. Lekatsas et. al. [LHW00] realized that assigning Huffman codes to whole instruction was less efficient than assigning codes to the different parts of an instruction (opcodes, registers, conditionals) by separate. Benini et.al. [BMMP99] store the code compressed and then rely on a hardware unit to decompress it on the fly, prior to passing it to the CPU. This method is also present with variations in a number of different papers [ACAP00]. Lefurgy et al. [LPM00] reviewed this strategy later by proposing a decompression phase that uses a special unit of the CPU.

Compression Using Semantics Primer relies heavily on the semantics of programs to perform compression. This intuition has been explored before by Frazer [Fra99] who proposes to discover common patterns in code using Decision Trees. Frazer's approach however, was designed to be used on internal representations of programs, requiring a compilation phase after decompression or an interpreter.

Code Compaction Several techniques fall under the umbrella name of Code Compaction [DEMDS00, DBKC$^+$03, ERBS16, RSSR16, HTP$^+$07]. These techniques does not compress the code. Rather, they work by removing unreachable or similar code. For example, [CM99] and then [DEMDS00] propose a compiler optimization that search for related areas of code. When two or more similar code areas are found, one is kept and all others are replaced with a jump to the initially saved code segment. Post-pass compaction techniques [DBKC$^+$03] take this idea further by operating at the system level (i.e. not only similar parts in the program are factorized, but also on its dependencies or even in collections of programs). Romano et.al. [RSSR16] proposes a technique to detect methods that are never used in a Java program to reduce system's size. Traditional compiler optimizations such Dead Code Elimination and Invariant Code also reduce code size.

Compaction techniques are related to Primer because they do not preserve the original program being optimized. However, all compaction techniques perform sound transformations, while Primer relies on unsound transformations. Also, they do not transforms the code into another representation, therefore they can be used in conjunction with a compression algorithm such as Primer.

5.5 Conclusions

This chapter presented Primer, the first (to the best of our knowledge) lossy code compression algorithm. The chapter described the algorithm's inner working, both the Heuristic Ranking function and the Lossy Phase. Also, the results of our experiments were given showing the effectiveness of Heuristic Ranking to identify the original candidate and the Lossy Phase's ability to further reduce the compression ratio without affecting the compressed program's QoS. Compared with `GZip`, Premier was able to outperform the industry-standard algorithm by 10%.

Chapter 6

Conclusions and Perspectives

6.1 Conclusions

Approximate Computing is a broad field of study that propose a new point of view on accuracy. Instead of preserving accuracy at all cost, practitioners of the field realized that many applications such as Image Processing, Machine Learning and Big Data can endure some degree of imprecision while still producing good enough results. This opens a world of new opportunities to reduce energy consumption, execution times, circuit area and storage space, among others. The ideas of Approximate Computing have found application in diverse fields such as embedded devices, high performance computing and media processing.

This thesis had proposed three contributions to the field of approximate computing (i) Approximate Unrolling: a machine-independent compiler optimization, (ii) AutoJMH: a tool to measure speedup of approximate pieces of code and (iii) Primer: the first lossy algorithm for ARM instructions.

Approximate Unrolling exploits the data locality found in time series to interpolate costly computations and reduce the execution times and energy consumption of loops mapping values to an array. By using interpolation, the optimization reduces execution times and energy consumption of transformed loops as hypothesized initially. Also, interpolations improved the accuracy and robustness obtained when transforming loops with Loop Perforation, the current state-of-the-art approximate loop optimization. Our implementation of Approximate Unrolling in the OpenJDK C2 compiler was able to estimate with high confidence if a forgiving zone would indeed reduce execution times by means of a prediction policy. The current implementation is still unable to determine if by applying the optimization accuracy would drop to unacceptable levels. Good solutions for this particular problem exists in the form of annotations or alter-and-test dynamic analysis. During our experiments we decided to use annotations but other methods could be used also since Approximate Unrolling is orthogonal to the concern of identifying forgiving zones.

AutoJHM creates payloads for JMH microbenchmarks in such a way that common mistakes made by engineers without experience designing this kind of tests are avoided. Microbenchmarks allows to observe smaller differences in performance way smaller than those observed by benchmarks. Yet, microbenchmarks are notoriously difficult to design properly. With AutoJMH, the amount of technical knowledge needed to design microbenchmarks properly is reduced. This allows more engineers to use these tests, which present unique advantages over benchmarks and profilers. AutoJMH's evaluation showed its ability to outperform engineers without experience and to match experienced performance engineers. Certainly, AutoJMH is able to avoid only a small set of mistakes such as tests prone to Dead Code, Constant Folding and unstable states. Yet, those errors were shown to be among the most commonly made by inexperienced designers. Our results demonstrated that microbenchmarking can be automated to a large extent and hint that the process can be fully automated.

Primer is the first lossy compression scheme for ARM instructions. The algorithm treats a program's forgiving regions as lossy data, allowing it to outperform the compression ratio of the current state-of-the-art code compression schemes. Unfortunately, the lossy nature of the algorithm can only be applied in programs with forgiving zones. The idea is still in its infancy, better compression ratios could be obtained if with further research in this direction where to be done. Also, a proper assessment of the compression times and energy consumption of the algorithm is required.

6.2 Perspectives

This sections describes a potential roadmap for the continuation of all the work presented in this thesis. Initially, the section recommends several tasks to improve the presented techniques and tools. Finally, some steps to achieve industrial application of our results are proposed.

6.2.1 Improving Existing Results

Predict Accuracy Losses The current implementation of Approximate Unrolling is able to predict which loops will obtain speedups benefits from the transformations. This is done by means of a policy. Yet, the current policy is unable to predict whether the accuracy drops in loops will be acceptable after being optimized. A way to predict accuracy losses in optimized loops is then needed. We envision this to be done in two ways, by static analysis or manually by a programmer.

The static analysis would try to find loops abiding to the patterns found in Section 3.4.4. These loops are the ones where the function being mapped is a smooth function. Smooth functions are defined as those for which their derivatives exists up to a certain order. While is not feasible to tests this condition for all possible functions without a considerable amount of computation, it can be estimated with ease for some simpler functions composed by arithmetic computations and intrinsic (i.e. built-in) functions such as `log` or `sin`.

Besides an automated static analysis, another option to identify Approximate Unrolling amenable zones is for a programmer to do so manually. However, existing approximate languages do not consider having more than one technique that can be applied in the same type of forgiving zone (i.e. Approximate Unrolling and Loop Perforation). Therefore, if more than one technique can be applied, no approximate language features a way to hint the compiler regarding which approximation is best to use. Another shortcoming of current languages is that some allow declaring approximable variables [SDF+11a], some to specify acceptable levels of approximation [MCA+14, BSGC15] and some to express conditions on which the system should change [HBZ+16, BC10, SKG+07]. Yet, no existing language binds all these concerns together. These shortcomings indicates that the field of approximate languages is far from mature and that new developments should be made in order to propose tools that can be used outside the lab.

Towards Fully-Automated Generation of Microbenchmarks While AutoJMH is able to avoid several common mistakes made by engineers without experience in microbenchmarking, is still far from being able to fully automate all the potential situations that can arise during the design of a tests. These include handling the situations that can arise in multi-threaded applications such as false-sharing or asymmetric loads. Another feature is to analyze the data passed into the tests, as it can cause corner cases of branch prediction by the CPU. Another improvement that should be added to the tool is the analysis of the tests' output results to detect patterns that could signal a potential badly constructed payload, such as a suspiciously fast-executing microbenchmarks. These are some examples of the situations that should be addressed in the next version of AutoJMH.

Improving Primer While Primer is able to provide state-of-the-art compression ratios, we believe there is still much to be explored in this new direction. The Primer's lossy schema can be improved. Currently, the lossy algorithm follows the lossless and tries to improve upon it. However, much knowledge is being obtained regarding resilience, diversity and unsound transformations in programs by the community. Therefore, we believe better schemes to exploit the newly discover lossy nature of code remain to be discovered. Those algorithms should focus on the newly found nature of code, just like the JPEG or MPEG standards do in media applications.

6.2.2 Technology Transfer Into Industry

Approximate Unrolling The idea for Approximate Unrolling came initially as a way to support the implementation of software synthesizers in low-energy embedded devices. The grand picture is to create a programmable smart sound card that could process sound in real time. In live sound effects processing, real-time constraints receive higher priority than sound quality. Hence, Approximate Computing proposes an attractive alternative to enforce real-time constraints.

Our future work will consist in transferring the gained knowledge during the thesis into real-time processing units, where Approximate Computing will be used to enforce real time requirements. Most software sound processing units consist in a master loop that maps the results of a function into a buffer. Approximate Unrolling is certainly a viable technique in this contexts, as is able to notably increase the performance of loops processing sound while maintaining sound quality reduction acceptable or even unnoticeable.

AutoJMH In order to make AutoJMH an industry-ready tool three main aspects must be addressed (i) usability (ii) test generation execution time and (iii) constant maintenance. A first step will be to receive user feedback regarding the tool's usage and then adapt the tool responding to this feedback. Later on, improvements to reduce the time the analysis takes to run would be needed, as microbenchmarks are used for small constant improvements and tests should be generated as quickly as possible. Finally, the developers of AutoJMH should constantly adapt the tool to follow the inner workings of the Java VM , as many of the design choices made depends on the set of optimizations made by the Java VM.

Primer During the presentation of Primer in this thesis, issues such as speed of compression (i.e. the time Primer takes to execute) and the energy consumption of the algorithm where left unexplored. These are important concerns, since the main point of compression is precisely to reduce the energy required to receive the firmware by a given node. Some of the questions we have received in early feedback when presenting the technique to industry professionals is whether it is possible to actually implement Heuristic Ranking in low-energy devices or whether the actual energy consumption of the algorithm will justify its usage. Hence, in order to obtain an industry-ready technique, a careful analysis on performance and energy consumption of Primer should be performed in the very near future.

Bibliography

[AAFM10] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance Analysis of Idle Programs. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '10, pages 739–753, New York, NY, USA, 2010. ACM.

[AAFR10] C. Alippi, G. Anastasi, M. Di Francesco, and M. Roveri. An Adaptive Sampling Algorithm for Effective Energy Management in Wireless Sensor Networks With Energy-Hungry Sensors. IEEE Transactions on Instrumentation and Measurement, 59(2):335–344, February 2010.

[ABCF11] G. Agosta, M. Bessi, E. Capra, and C. Francalanci. Dynamic memoization for energy efficiency in financial applications. In 2011 International Green Computing Conference and Workshops, pages 1–8, July 2011.

[ACAP00] G. Araujo, P. Centoducatte, R. Azevedo, and R. Pannain. Expression-tree-based algorithms for code compression on embedded RISC architectures. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 8(5):530–533, October 2000.

[ACDFP09] Giuseppe Anastasi, Marco Conti, Mario Di Francesco, and Andrea Passarella. Energy conservation in wireless sensor networks: A survey. Ad Hoc Networks, 7(3):537–568, May 2009.

[ACV05] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy Memoization for Floating-Point Multimedia Applications. IEEE Transactions on Computers, 54(7), 2005.

[ACW$^+$09] Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. In ACM Conference on Programming Language Design and Implementation (PLDI), 2009.

[AJ09] Alex Buckley and John Rose. Oow 2009 Towards A Universal Vm, 2009.

[AKPW83] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of Control Dependence to Data Dependence. In Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[Ale13a] Aleksey Shipilev. Java Microbenchmarks Harness (the lesser of two evils). Belgium, 2013.

[Ale13b] Aleksey Shipilev. Necessar(il)y Evil dealing with benchmarks, ugh, July 2013.

[Ale14a] Aleksey Shipilev. Java Benchmarking as easy as two timestamps, July 2014.

[Ale14b] Aleksey Shipilev. Java Benchmarking, Timestamping Failures, October 2014.

[Ale15] Aleksey Shipilev. The Black Magic of (Java) Method Dispatch. `http://shipilev.net/blog/2015/black-magic-method-dispatch/`, 2015.

[BAM14] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored Source Code Transformations to Synthesize Computationally Diverse Program Variants. arXiv:1401.7635 [cs], pages 149–159, 2014.

[BARCM15a] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio, and Martin Monperrus. Automatic Software Diversity in the Light of Test Suites. arXiv:1509.00144 [cs], September 2015.

[BARCM15b] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio, and Martin Monperrus. DSpot: Test Amplification for Automatic Assessment of Computational Diversity. arXiv:1503.05807 [cs], March 2015.

[BBS+03] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield. New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors. IBM Journal of Research and Development, 47(5.6):653–670, September 2003.

[BC10] Woongki Baek and Trishul M. Chilimbi. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[BFG+03] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of Code-size Reduction Methods. ACM Comput. Surv., 35(3):223–267, September 2003.

[BJS09] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: Automated Test Generation for Worst-case Complexity. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 463–473, Washington, DC, USA, 2009. IEEE Computer Society.

[BM99] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In Proceedings Fifth International Symposium on High-Performance Computer Architecture, pages 13–22, January 1999.

[BM15] Benoit Baudry and Martin Monperrus. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. ACM Comput. Surv., 48(1):16:1–16:26, September 2015.

[BMM14] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<T>: A First-order Type for Uncertain Data. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, pages 51–66, New York, NY, USA, 2014. ACM.

[BMMP99] L. Benini, A. Macii, E. Macii, and M. Poncino. Selective instruction compression for memory energy reduction in embedded systems. In Proceedings. 1999 International Symposium on Low Power Electronics and Design (Cat. No.99TH8477), pages 206–211, August 1999.

[Bri05] Brian Goets. Java theory and practice: Anatomy of a flawed microbenchmark. `http://www.ibm.com/developerworks/library/j-jtp02225/`, February 2005.

[BS13] Stephen Brown and Cormac J. Sreenan. Software Updating in Wireless Sensor Networks: A Survey and Lacunae. Journal of Sensor and Actuator Networks, 2(4):717–760, November 2013.

[BSGC15] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability Type Inference for Flexible Approximate Programming. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, pages 470–487, New York, NY, USA, 2015. ACM.

[Bur98] P Burk. JSyn - A Real-time Synthesis API for Java. In Proceedings of the International Computer Music Conference, pages 252–255. International Computer Music Association, 1998.

[CB13] Charlie Curtsinger and Emery D. Berger. STABILIZER: Statistically Sound Performance Evaluation. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 219–228, New York, NY, USA, 2013. ACM.

[CCRR13] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pages 1–9, May 2013.

[CDHH06] David Chu, Amol Deshpande, Joseph M. Hellerstein, and Wei Hong. Approximate Data Collection in Sensor Networks Using Probabilistic Models. In Proceedings of the 22Nd International Conference on Data

Engineering, ICDE '06, pages 48–, Washington, DC, USA, 2006. IEEE Computer Society.

[CG11] Jason Cong and Karthik Gururaj. Assuring application-level correctness against soft errors. In IEEE–ACM International Conference on Computer-Aided Design (ICCAD), 2011.

[Cli95] Cliff Click. Global Code Motion/Global Value Numbering. In Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation, PLDI '95, pages 246–257, New York, NY, USA, 1995. ACM.

[Cli10] Cliff Click. The Art of Java Benchmarking, 2010.

[CM99] Keith D. Cooper and Nathaniel McIntosh. Enhanced Code Compression for Embedded RISC Processors. In Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI '99, pages 139–149, New York, NY, USA, 1999. ACM.

[CP95] Cliff Click and Michael Paleczny. A Simple Graph-based Intermediate Representation. In Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations, IR '95, pages 35–49, New York, NY, USA, 1995. ACM.

[CT05] Keith D. Cooper and Linda Torczon. Engineering a Compiler. Morgan Kaufmann, 2005.

[CYQ$^+$16] Y. Chen, X. Yang, F. Qiao, J. Han, Q. Wei, and H. Yang. A Multi-accuracy-Level Approximate Memory Architecture Based on Data Significance Analysis. In 2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pages 385–390, July 2016.

[CZSL12] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy Types. In ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2012.

[Dav03] David Astels. Test-Driven Development: A Practical Guide: A Practical Guide. Prentice Hall, Upper Saddle River, N.J.; London, 1 edition edition, July 2003.

[DBKC$^+$03] Bruno De Bus, Daniel Kästner, Dominique Chanet, Ludo Van Put, and Bjorn De Sutter. Post-pass Compaction Techniques. Commun. ACM, 46(8):41–46, August 2003.

[Del15] Pierre Delforge. America's Data Centers Consuming and Wasting Growing Amounts of Energy. `https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy`, February 2015.

[DEMDS00] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler Techniques for Code Compaction. ACM Trans. Program. Lang. Syst., 22(2):378–415, March 2000.

[Dmi15] Dmitry Vyazalenko. Using JMH in a real world project. Moscow, October 2015.

[DNT16] S. Dutt, S. Nandi, and G. Trivedi. A comparative survey of approximate adders. In 2016 26th International Conference Radioelektronika (RADIOELEKTRONIKA), pages 61–65, April 2016.

[dP11] Florent de Dinechin and Bogdan Pasca. Designing Custom Arithmetic Data Paths with FloPoCo. IEEE Des. Test, 28(4):18–27, July 2011.

[DVM12] K. Du, P. Varman, and K. Mohanram. High performance reliable variable latency carry select addition. In 2012 Design, Automation Test in Europe Conference Exhibition (DATE), pages 1257–1262, March 2012.

[ENN13] Hadi Esmaeilzadeh, Kangqi Ni, and Mayur Naik. Expectation-Oriented Framework for Automating Approximate Programming. Technical Report GT-CS-13-07, Georgia Institute of Technology, 2013.

[ERBS16] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. Extracting Equivalent SQL from Imperative Code in Database Applications. In Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16, pages 1781–1796, New York, NY, USA, 2016. ACM.

[ERZJ14] Schuyler Eldridge, Florian Raudies, David Zou, and Ajay Joshi. Neural Network-based Accelerators for Transcendental Function Approximation. In Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI, GLSVLSI '14, pages 169–174, New York, NY, USA, 2014. ACM.

[ESCB12] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In IEEE/ACM International Symposium on Microarchitecture (MICRO), 2012.

[FLL12] Yuntan Fang, Huawei Li, and Xiaowei Li. SoftPCM: Enhancing Energy Efficiency and Lifetime of Phase Change Memory in Video Applications via Approximate Write. In Asian Test Symposium (ATS), 2012.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. ACM Trans. Program. Lang. Syst., 9(3):319–349, July 1987.

[Fra99] Christopher W. Fraser. Automatic Inference of Models for Statistical Code Compression. In Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, PLDI '99, pages 242–246, New York, NY, USA, 1999. ACM.

[FRPVTCS13] A. Falcón-Ruiz, J. E. Paz-Viera, A. Taboada-Crispí, and H. Sahli. Automatic Bound Estimation for JPEG 2000 Compressing Leukocytes Images. In V Latin American Congress on Biomedical Engineering CLAIB 2011 May 16-21, 2011, Habana, Cuba, IFMBE Proceedings, pages 547–550. Springer, Berlin, Heidelberg, 2013.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[GBNN15] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, pages 383–397, New York, NY, USA, 2015. ACM.

[GCC17] GCC Developers. Optimize Options - Using the GNU Compiler Collection (GCC). `https://gcc.gnu.org/onlinedocs/gcc-4.6.2/gcc/Optimize-Options.html`, 2017.

[GEB08] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java Performance Evaluation Through Rigorous Replay Compilation. In Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, OOPSLA '08, pages 367–384, New York, NY, USA, 2008. ACM.

[GFX12] Mark Grechanik, Chen Fu, and Qing Xie. Automatically Finding Performance Problems with Feedback-directed Learning Software Testing. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 156–166, Piscataway, NJ, USA, 2012. IEEE Press.

[GLY07] B. Gedik, L. Liu, and P. S. Yu. ASAP: An Adaptive Sampling Approach to Data Collection in Sensor Networks. IEEE Transactions on Parallel and Distributed Systems, 18(12):1766–1783, December 2007.

[GMP$^+$11] V. Gupta, D. Mohapatra, Sang Phill Park, A. Raghunathan, and K. Roy. IMPACT: Imprecise adders for low-power approximate computing. In International Symposium on Low Power Electronics and Design (ISLPED), 2011.

[GPI06] S. Goel, A. Passarella, and T. Imielinski. Using buddies to live longer in a boring world [sensor network protocol]. In Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06), pages 5 pp.–346, March 2006.

[GR11] John Gantz and David Reinsel. Extracting Value from Chaos. Technical report, June 2011.

[GR14] Beayna Grigorian and Glenn Reinman. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In NASA–ESA Conference On Adaptive Hardware And Systems (AHS), 2014.

[GRE$^+$01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), pages 3–14, December 2001.

[Hai17] Li Haifeng. Smile - Statistical Machine Intelligence and Learning Engine. `https://github.com/haifengl/smile`, 2017.

[Has] Trevor Hastie. The Elements of Statistical Learning - Data Mining, Inference. Springer.

[HBZ$^+$16] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. Disciplined Inconsistency with Consistency Types. In Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16, pages 279–293, New York, NY, USA, 2016. ACM.

[HDG$^+$12] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance Debugging in the Large via Mining Millions of Stack Traces. In Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pages 145–155, Piscataway, NJ, USA, 2012. IEEE Press.

[HLM$^+$15] Vojtěch Horký, Peter Libič, Lukáš Marek, Antonin Steinhauser, and Petr T\r uma. Utilizing Performance Unit Tests To Increase Performance Awareness. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ICPE '15, pages 289–300, New York, NY, USA, 2015. ACM.

[HLST15] Vojtěch Horký, Peter Libič, Antonin Steinhauser, and Petr T\r uma. DOs and DON'Ts of Conducting Performance Measurements in Java.

In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ICPE '15, pages 337–340, New York, NY, USA, 2015. ACM.

[HMS$^+$12] Andreas Heinig, Vincent J. Mooney, Florian Schmoll, Peter Marwedel, Krishna Palem, and Michael Engel. Classification-Based Improvement of Application Robustness and Quality of Service in Probabilistic Computer Systems. In Architecture of Computing Systems – ARCS 2012, Lecture Notes in Computer Science, pages 1–12. Springer, Berlin, Heidelberg, February 2012.

[HO13] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In 2013 18th IEEE European Test Symposium (ETS), pages 1–6, May 2013.

[HSC$^+$11] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic Knobs for Responsive Power-aware Computing. In Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, pages 199–212, New York, NY, USA, 2011. ACM.

[HSD11] Jonathon S. Hare, Sina Samangooei, and David P. Dupplaw. OpenIMAJ and ImageTerrier: Java libraries and tools for scalable multimedia analysis and indexing of images. In Proceedings of the 19th ACM International Conference on Multimedia, MM '11, pages 691–694, New York, NY, USA, 2011. ACM.

[HTP$^+$07] Haifeng He, John Trimble, Somu Perianayagam, Saumya Debray, and Gregory Andrews. Code Compaction of an Operating System Kernel. In Proceedings of the International Symposium on Code Generation and Optimization, CGO '07, pages 283–298, Washington, DC, USA, 2007. IEEE Computer Society.

[Jac14] Jacob Bramley. Arm Community. `https://community.arm.com/processors/b/blog/posts/condition-codes-1-condition-flags-and-codes`, 2014.

[JBLF10] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. DFS: A File System for Virtualized Flash Storage. Trans. Storage, 6(3):14:1–14:25, September 2010.

[JC04] Jaein Jeong and D. Culler. Incremental network programming for wireless sensors. In 2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004., pages 25–33, October 2004.

[JLL$^+$17] Honglan Jiang, Cong Liu, Leibo Liu, Fabrizio Lombardi, and Jie Han. A Review, Classification, and Comparative Evaluation of Approximate Arithmetic Circuits. J. Emerg. Technol. Comput. Syst., 13(4):60:1–60:34, August 2017.

[JLM$^+$16] H. Jiang, C. Liu, N. Maheshwari, F. Lombardi, and J. Han. A comparative evaluation of approximate multipliers. In 2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH), pages 191–196, July 2016.

[Jul14a] Julien Ponge. Revisiting a (JSON) Benchmark, September 2014.

[Jul14b] Julien Ponge. Avoiding Benchmarking Pitfalls on the JVM. Oracle Java Magazine, July/August 2014.

[JZA$^+$05] Hyewon Jun, Wenrui Zhao, M. H. Ammar, E. W. Zegura, and Chungki Lee. Trading latency for energy in wireless ad hoc networks using message ferrying. In Third IEEE International Conference on Pervasive Computing and Communications Workshops, pages 220–225, March 2005.

[KOR] Michael Kuperberg, Fouad Omri, and Ralf Reussner. Automated Benchmarking of Java APIs.

[KZSM15] Daya S. Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. Rumba: An Online Quality Management System for Approximate Computing. In International Symposium on Computer Architecture (ISCA), 2015.

[LC02] Philip Levis and David Culler. MatÉ: A Tiny Virtual Machine for Sensor Networks. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS X, pages 85–95, New York, NY, USA, 2002. ACM.

[LDK99] Stan Liao, Srinivas Devadas, and Kurt Keutzer. A Text-compression-based Method for Code Size Minimization in Embedded Systems. ACM Trans. Des. Autom. Electron. Syst., 4(1):12–38, January 1999.

[LEN$^+$11] Avinash Lingamneni, Christian C. Enz, Jean-Luc Nagel, Krishna V. Palem, and Christian Piguet. Energy parsimonious circuit design through probabilistic pruning. In Design, Automation and Test in Europe (DATE), 2011.

[LGI09] Erietta Liarou, Romulo Goncalves, and Stratos Idreos. Exploiting the Power of Relational Databases for Efficient Stream Processing. In Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '09, pages 323–334, New York, NY, USA, 2009. ACM.

[LHW00] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. Code Compression for Low Power Embedded System Design. In Proceedings of the 37th Annual Design Automation Conference, DAC '00, pages 294–299, New York, NY, USA, 2000. ACM.

[LPM00] C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. In Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550), pages 218–228, 2000.

[LPYD15] Kenan Liu, Gustavo Pinto, and Lu Yu David. Data-Oriented Characterization of Application-Level Energy Optimization. 2015.

[LW06] H. Lekatsas and W. Wolf. SAMC: A Code Compression Algorithm for Embedded Processors. Trans. Comp.-Aided Des. Integ. Cir. Sys., 18(12):1689–1701, November 2006.

[MAMJ15] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. Doppelganger: A cache for approximate computing. In 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 50–61, December 2015.

[MBEJ14] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load Value Approximation. In IEEE/ACM International Symposium on Microarchitecture (MICRO), 2014.

[MCA$^+$14] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. pages 309–328. ACM Press, 2014.

[MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[MGO14] Jin Miao, Andreas Gerstlauer, and Michael Orshansky. Multi-level Approximate Logic Synthesis Under General Error Constraints. In Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design, ICCAD '14, pages 504–510, Piscataway, NJ, USA, 2014. IEEE Press.

[MHG10] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. Lucene in Action, Second Edition: Covers Apache Lucene 3.0. Manning Publications Co., Greenwich, CT, USA, 2010.

[Mit16] Sparsh Mittal. A Survey of Techniques for Approximate Computing. ACM Comput. Surv., 48(4):62:1–62:33, March 2016.

[MRCB10] Jiayuan Meng, Anand Raghunathan, Srimat Chakradhar, and Surendra Byna. Exploiting the forgiving nature of applications for scalable parallel execution. In IEEE International Parallel & Distributed Processing Symposium, 2010.

[MSHR10] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of Service Profiling. In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 25–34, New York, NY, USA, 2010. ACM.

[Naj94] F. N. Najm. A survey of power estimation techniques in VLSI circuits. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2(4):446–455, December 1994.

[NCRL15] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, pages 902–912, Piscataway, NJ, USA, 2015. IEEE Press.

[NNR09] Naveen Muralimanohar, Norman P. Jouppi, and Rajeev Balasubramonian. CACTI 6.0: A Tool to Model Large Caches. Technical report, HP Laboratories, April 2009.

[NRRW11] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. arXiv:1106.5730 [cs, math], June 2011.

[NZMD15] Ankit P. Navik, Mukesh A. Zaveri, Sristi Vns Murthy, and Manoj Dawarwadikar. Microbenchmark Based Performance Evaluation of GPU Rendering. In N. R. Shetty, N. H. Prasad, and N. Nalini, editors, Emerging Research in Computing, Information, Communication and Applications, pages 407–415. Springer India, 2015.

[ODL15] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pages 369–378, New York, NY, USA, 2015. ACM.

[PEZ$^+$15] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. FlexJava: Language Support for Safe and Modular Approximate Programming. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 745–757, New York, NY, USA, 2015. ACM.

[PHG14] Michael Pradel, Markus Huggler, and Thomas R. Gross. Performance Regression Testing of Concurrent Classes. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pages 13–25, New York, NY, USA, 2014. ACM.

[RCS$^+$11] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[RGNN$^+$13] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning Assistant for Floating-point Precision. In International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2013.

[RMGB13] A. Rahimi, A. Marongiu, R.K. Gupta, and L. Benini. A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters. In IEEE–ACM–IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013.

[RRV$^+$13] Nikola Rajovic, Alejandro Rico, James Vipond, Isaac Gelado, Nikola Puzovic, and Alex Ramirez. Experiences with Mobile Processors for Energy Efficient HPC. In Proceedings of the Conference on Design, Automation and Test in Europe, DATE '13, pages 464–468, San Jose, CA, USA, 2013. EDA Consortium.

[RRV$^+$14] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. ASLAN: Synthesis of Approximate Sequential Circuits. In Design, Automation and Test in Europe (DATE), 2014.

[RRWW14] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. ASAC: Automatic Sensitivity Analysis for Approximate Computing. In Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES '14, pages 95–104, New York, NY, USA, 2014. ACM.

[RSA$^+$15] Michael F. Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Monitoring and Debugging the Quality of Results in Approximate Programs. In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015.

[RSJR17] A. Raha, S. Sutar, H. Jayakumar, and V. Raghunathan. Quality Configurable Approximate DRAM. IEEE Transactions on Computers, 66(7):1172–1187, July 2017.

[RSNP12] Lakshminarayanan Renganarayanan, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. Programming with Relaxed Synchronization. In Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES), 2012.

[RSSR16] Simone Romano, Giuseppe Scanniello, Carlo Sartiani, and Michele Risi. A Graph-based Approach to Detect Unreachable Methods in Java Software. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, pages 1538–1541, New York, NY, USA, 2016. ACM.

[RW08] Pascal Rickenbach and Roger Wattenhofer. Decoding Code on a Sensor Node. In Proceedings of the 4th IEEE International Conference on Distributed Computing in Sensor Systems, DCOSS '08, pages 400–414, Berlin, Heidelberg, 2008. Springer-Verlag.

[SAHH15] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. A Low Latency Generic Accuracy Configurable Adder. In Design Automation Conference (DAC), 2015.

[Sam17] Adrian Sampson. Approximate Computing: An Annotated Bibliography, March 2017.

[ŠBD05] Marija Šalovarda, Ivan Bolkovac, and Hrvoje Domitrović. Estimating perceptual audio system quality using PEAQ algorithm. In ICECom (18; 2005), 2005.

[SBR+15] Adrian Sampson, André Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. Accept: A programmer-guided compiler framework for practical approximate computing. University of Washington Technical Report UW-CSE-15-01, 1, 2015.

[SDF+11a] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation — Full Proofs. Technical Report UW-CSE-10-12-01, University of Washington, 2011.

[SDF+11b] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[SDH$^+$14] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler Software Optimization for Reducing Energy. In International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014.

[SDMHR11] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[SJLM14] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based Approximation for Data Parallel Applications. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, pages 35–50, New York, NY, USA, 2014. ACM.

[SKG$^+$07] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A Language and Runtime System for Perpetual Systems. In ACM Conference on Embedded Networked Sensor Systems (SenSys), 2007.

[SLJ$^+$13] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. SAGE: Self-tuning Approximation for Graphics Engines. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, pages 13–24, New York, NY, USA, 2013. ACM.

[SLPG15] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating Performance Bottleneck Detection Using Search-based Application Profiling. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, pages 270–281, New York, NY, USA, 2015. ACM.

[SMR15] Phillip Stanley-Marbell and Martin Rinard. Lax: Driver Interfaces for Approximate Sensor Device Access. In 15th Workshop on Hot Topics in Operating Systems (HotOS XV), Kartause Ittingen, Switzerland, 2015. USENIX Association.

[SNSC13] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate Storage in Solid-state Memories. In IEEE/ACM International Symposium on Microarchitecture (MICRO), 2013.

[SPM$^+$14] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Expressing and Verifying

Probabilistic Assertions. In ACM Conference on Programming Language Design and Implementation (PLDI), 2014.

[SV06] Mario Strasser and Harald Vogt. Autonomous and Distributed Node Recovery in Wireless Sensor Networks. In Proceedings of the Fourth ACM Workshop on Security of Ad Hoc and Sensor Networks, SASN '06, pages 113–122, New York, NY, USA, 2006. ACM.

[SV15] Michael J. Steindorfer and Jurgen J. Vinju. Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, pages 783–800, New York, NY, USA, 2015. ACM.

[SZ02] Y. Q. Shi and Xi Min Zhang. A new two-dimensional interleaving technique using successive packing. IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 49(6):779–789, June 2002.

[Tar] Robert Tarjan. Testing flow graph reducibility. `http://dl.acm.org/citation.cfm?id=804040`.

[TC11] Linda Torczon and Keith Cooper. Engineering A Compiler. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[Thr00] S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), volume 1, pages 306–312 vol.1, 2000.

[Tip94] Frank Tip. A Survey of Program Slicing Techniques. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1994.

[TNR00] Jonathan Ying Fai Tong, David Nagle, and Rob. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 8(3), 2000.

[TPY$^+$14] Bradley Thwaites, Gennady Pekhimenko, Amir Yazdanbakhsh, Jongse Park, Girish Mururu, Hadi Esmaeilzadeh, Onur Mutlu, and Todd Mowry. Rollback-Free Value Prediction with Approximate Loads. In International Conference on Parallel Architectures and Compilation Techniques (PACT), 2014.

[TZW+15] Ye Tian, Qian Zhang, Ting Wang, Feng Yuan, and Qiang Xu. ApproxMA: Approximate Memory Access for Dynamic Precision Scaling. In Proceedings of the 25th Edition on Great Lakes Symposium on VLSI, GLSVLSI '15, pages 337–342, New York, NY, USA, 2015. ACM.

[Var17] Various. ARMv5 Architecture Reference Manual. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html`, 2017.

[VCC+13] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality Programmable Vector Processors for Approximate Computing. In IEEE/ACM International Symposium on Microarchitecture (MICRO), 2013.

[Vic94] Christopher Allen Vick. SSA-Based Reduction of Operator Strength. Thesis, Rice University, 1994.

[VPC+15] Vassilis Vassiliadis, Konstantinos Parasyris, Charalambos Chalios, Christos D. Antonopoulos, Spyros Lalis, Nikolaos Bellas, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. A Programming Model and Runtime System for Significance-aware Energy-efficient Computing. In Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pages 275–276, New York, NY, USA, 2015. ACM.

[VRLS15] Swagath Venkataramani, Anand Raghunathan, Jie Liu, and Mohammed Shoaib. Scalable-effort Classifiers for Energy-efficient Machine Learning. In Design Automation Conference (DAC), 2015.

[VRRR14] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. AxNN: Energy-efficient Neuromorphic Systems Using Approximate Computing. In Proceedings of the 2014 International Symposium on Low Power Electronics and Design, ISLPED '14, pages 27–32, New York, NY, USA, 2014. ACM.

[VSK+12] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. SALSA: Systematic Logic Synthesis of Approximate Circuits. In Design Automation Conference (DAC), 2012.

[WBMP05] Z. M. Wang, S. Basagni, E. Melachrinoudis, and C. Petrioli. Exploiting Sink Mobility for Maximizing Sensor Networks Lifetime. In Proceedings of the 38th Annual Hawaii International Conference on System Sciences, pages 287a–287a, January 2005.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value Dependence Graphs: Representation Without Taxation. In Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’94, pages 297–310, New York, NY, USA, 1994. ACM.

[Wil93] William B. Pennebaker. JPEG - Still Image Data Compression Standard. Springer, 1993.

[WS14] Lucas Wanner and Mani Srivastava. ViRUS: Virtual Function Replacement Under Stress. In USENIX Workshop on Power-Aware Computing and Systems (HotPower), 2014.

[XMK16] Q. Xu, T. Mytkowicz, and N. S. Kim. Approximate Computing: A Survey. IEEE Design Test, 33(1):8–22, February 2016.

[YFE$^+$07] Thomas Y. Yeh, Petros Faloutsos, Milos Ercegovac, Sanjay J. Patel, and Glen Reinman. The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration. In IEEE/ACM International Symposium on Microarchitecture (MICRO), 2007.

[YMT$^+$15] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, Jongse Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan. Axilog: Language support for approximate hardware design. In Design, Automation and Test in Europe (DATE), 2015.

[YTE$^+$16] A. Yazdanbakhsh, B. Thwaites, H. Esmaeilzadeh, G. Pekhimenko, O. Mutlu, and T. C. Mowry. Mitigating the Memory Bottleneck With Approximate Load Value Prediction. IEEE Design Test, 33(1):32–42, February 2016.

[YWY$^+$13] Rong Ye, Ting Wang, Feng Yuan, Rakesh Kumar, and Qiang Xu. On Reconfiguration-oriented Approximate Adder Design and Its Application. In IEEE–ACM International Conference on Computer-Aided Design (ICCAD), 2013.

[ZED11] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. Automatic Generation of Load Tests. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE ’11, pages 43–52, Washington, DC, USA, 2011. IEEE Computer Society.

[ZK06] Ahmad Zmily and Christos Kozyrakis. Simultaneously Improving Code Size, Performance, and Energy in Embedded Processors. In Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings, DATE ’06, pages 224–229, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[ZPL14] Hang Zhang, Mateja Putic, and John Lach. Low Power GPGPU Computation with Imprecise Hardware. In Proceedings of the 51st Annual

Design Automation Conference, DAC '14, pages 99:1–99:6, New York, NY, USA, 2014. ACM.

List of Figures

Abstract

Approximate Computing is based on the idea that significant improvements in CPU, energy and memory usage can be achieved when small levels of inaccuracy can be tolerated. This is an attractive concept, since the lack of resources is a constant problem in almost all computer science domains. From large super-computers processing today's social media big data, to small, energy-constraint embedded systems, there is always the need to optimize the consumption of some scarce resource. Approximate Computing proposes an alternative to this scarcity, introducing accuracy as yet another resource that can be in turn traded by performance, energy consumption or storage space.

The first part of this thesis proposes the following two contributions to the field of Approximate Computing:

- Approximate Loop Unrolling: a compiler optimization that exploits the approximative nature of signal and time series data to decrease execution times and energy consumption of loops processing it. Our experiments showed that the optimization increases considerably the performance and energy efficiency of the optimized loops (150% - 200%) while preserving accuracy to acceptable levels.
- Primer: the first ever lossy compression algorithm for assembler instructions, which profits from programs' forgiving zones to obtain a compression ratio that outperforms the current state-of-the-art up to a 10%.

The main goal of Approximate Computing is to improve the usage of resources such as performance or energy. Therefore, a fair deal of effort is dedicated to observe the actual benefit obtained by exploiting a given technique under study. One of the resources that have been historically challenging to accurately measure is execution time. Hence, the second part of this thesis proposes the following tool:

- AutoJMH: a tool to automatically create performance microbenchmarks in Java. Microbenchmarks provide the finest grain performance assessment. Yet, requiring a great deal of expertise, they remain a craft of a few performance engineers. The tool allows (thanks to automation) the adoption of microbenchmark by non-experts. Our results shows that the generated microbencharks match the quality of payloads handwritten by performance experts and outperforms those written by professional Java developers without experience in microbenchmarking.