FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Suggesting Loop Unrolling Using a Heuristic-guided Approach

Pedro Pinto

Master in Informatics and Computing Engineering

Supervisor: João Cardoso (Associate Professor)

July 30, 2012

Suggesting Loop Unrolling Using a Heuristic-guided Approach

Pedro Pinto

Master in Informatics and Computing Engineering

July 30, 2012

Abstract

The development of embedded applications typically faces memory and/or execution time constraints. In order to improve performance advanced compilers may use code transformations. There are cases where code transformations are not automatically applied and it is up to the developer to manually apply them. One of the most relevant code transformations is Loop Unrolling. It is a widely studied transformation and it is able to improve the performance of many loops. It is easily implemented and its applicability is always legal. The main goal of this dissertation is to propose an approach to help developers decide about the application of Loop Unrolling and about the unroll factor to use. We propose an approach that uses a set of heuristics, based on characteristics extracted from the source code of the loops being analyzed, to suggest loops for Loop Unrolling. These heuristics are based on characteristics that likely lead to performance gains when Loop Unrolling is applied, but also on characteristics that can make a loop inadequate for Loop Unrolling. The approach consists in software extensions of an existent source-to-source tool to extract features and on a software tool implementing the heuristics.

To validate and evaluate the proposed approach we developed a prototype that targets the PowerPC architecture, and more specifically the PowerPC 604 processor. For this processor, we created an instance of all the heuristics and metrics used. Using functions from real life applications and other kernels as benchmarks, we were able to determine how efficient is the proposed approach for predicting whether Loop Unrolling will have a positive or a negative impact on the loop performance. With these benchmarks we were also able to determine if our approach is capable of suggesting a suitable unroll factor for a loop that was considered a good Loop Unrolling candidate. Our software prototype was able to make correct suggestions for 75% of the 8 benchmarks evaluated. The unroll factor suggestion was close or the same as the optimal unroll factor in all the cases except one. Nevertheless, even in this case the difference on performance gain to the optimal unroll factor was not significant. The optimal unroll factor led to a performance improvement of 35.37% while the suggested unroll factor led to a performance improvement of 31.49%.

Resumo

O desenvolvimento de aplicações embebidas é geralmente confrontado com restrições de memória e/ou tempo de execução. De forma a conseguir melhorar o desempenho, compiladores especializados usam transformações de código. Existem casos onde não é possível aplicar estas transformações automaticamente e cabe ao programador aplicá-las manualmente. Uma das transformações de código mais relevantes é o Desenrolamento de Ciclos[1]. É uma transformação vastamente estudada e que é capaz de melhorar o desempenho de muitos ciclos. É facilmente implementada e a sua aplicação é sempre legal. O principal objetivo desta dissertação é propor uma abordagem para ajudar os programadores a decidir quando aplicar o Desenrolamento de Ciclos e qual o fator a utilizar. É proposta uma abordagem que usa um conjunto de heurísticas, que baseadas em características extraídas do código fonte dos ciclos a analisar, sugere quando e como aplicar esta transformação. Estas heurísticas são baseadas em características que é provável conduzirem a ganhos de desempenho quando o Desenrolamento de Ciclos é aplicado, mas também em características que podem fazer com que um ciclo seja considerado desadequado para esta transformação. A abordagem proposta consiste na extensão das funcionalidades de uma ferramenta *source-to-source* já existente para extrair características e numa ferramenta que implementa e testa as heurísticas.

De forma a conseguir validar e avaliar a abordagem proposta foi desenvolvido um protótipo que tem como alvo a arquitetura PowerPC, e mais especificamente o processador PowerPC 604. Foi criada uma instância para este processador de todas as heurísticas e métricas usadas. Usando funções de aplicações reais e outros exemplos como testes, foi possível determinar quão eficiente é a abordagem proposta a prever se o Desenrolamento de Ciclos terá um impacto positivo ou negativo no desempenho do ciclo. Com estes testes foi também possível determinar se a abordagem é capaz de sugerir um fator de desenrolamento adequado. O protótipo desenvolvido foi capaz de fazer sugestões corretas em 75% dos 8 testes avaliados. A sugestão do fator de desenrolamento foi bastante próxima ou igual ao fator ótimo em todos os casos menos um. No entanto, mesmo neste caso a diferença no ganho de desempenho obtido entre o fator ótimo e o fator sugerido não foi significativa. O fator ótimo trouxe uma melhoria de 35,37% enquanto que o fator sugerido trouxe uma melhoria de 31,49%.

[1]Do inglês *Loop Unrolling*.

Acknowledgements

First and foremost I would like to thank my family and especially my parents, without whom I would not be able to complete this thesis and my master's degree. Their support throughout the years and even more in this final semester, putting up with my bad mood and stress while always cheering me up, is invaluable.

I wish to thank my M.Sc. supervisor, João Cardoso, whose insightful guidance always kept me in the right direction. His help was of utmost importance during both the writing of this thesis and all the preparation that lead to it. Without his profound knowledge and sound advice this work would not be the same.

I also wish to thank everyone on the SPeCS group, with whom I shared a laboratory for these past months. I have to thank them for they helpful, relaxed nature that created a perfect work environment. Whenever needed, there was always a helping hand and a friendly word.

Finally, I wish to thank my M.Sc. colleagues. Having someone going through the same process and with who I could share most of my experience was a great deal and I feel it helped me a lot.

The author acknowledges REFLECT, a project funded by the European Community's Framework Programme 7 (FP7) under contract No. 248976, for allowing the use of reflectcc, a compiler part of the REFLECT design-flow and based on the CoSy compiler development system from ACE Associated Compiler Experts bv.

Pedro Pinto

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Performance is always an important factor, no matter what type of application we are dealing with. Nobody wants their laptop taking several minutes to open an image or their mobile phone running out of battery just for checking his email. While the average user, using a common, everyday system, might tolerate a few performance drops, there are applications where performance is a critical factor and should be accounted for during the development process. Embedded applications are an example of this, typically facing execution time, memory and energy constraints.

One way to improve performance is using code transformations. A number of code transformations are already used automatically by advanced compilers and optimization tools. Others are applied manually by developers. These transformations can target various parts of the program and can be applied at different levels (e.g., source code or intermediate compiler representation).

Programs spend most of their time executing a small portion of code [DJ01]. A well known rule of thumb is: *"A program executes about 90% of its instructions in 10% of its code."* [HP03]. Any transformation that is able to improve the performance of this 10% will likely lead to an overall improvement in performance. As it turns out, this small part of programs often corresponds to loops [VLCV01, Kob84]. This means that loop transformations are of critical importance when considering performance improvements.

There are many loop transformations, all of them with different goals and results. Next, we present some examples of loop transformations. **Loop Invariant Code Motion** moves code that is invariant inside the loop to outside the loop. Consider a two-level loop nest with an array access in the innermost loop whose subscript depends only on the outer loop induction variable. This access can be moved from the inner to the outer loop, eliminating $N \times (M - 1)$ array accesses, with N and M being the number of iterations of the outer and inner loops, respectively. **Loop Coalescing** combines nested loops into a single loop. This new loop uses only one induction variable which can lead to less overhead and better load balancing. However, the two indexes that were previously used still need to be calculated from this new induction variable. This transformation can improve performance, only if these calculations can be simplified. Figure 1.1 shows an example of this

```c
for(i = 0; i < N; i++)
   for(j = 0; j < M; j++)
      array[i][j] = array[i][j] + c;
```

(a) Before

```c
int maxVal = N * M;

for( ij=0 ; ij<maxVal ; ij++ )
{
   i = ij / M;
   j = ij % M;
   array[i][j] = array[i][j] + c;
}
```

(b) After

Figure 1.1: An example of Loop Coalescing. The two indexes are still calculated using the new induction variable.

transformation. **Loop Collapsing** is a very similar transformation as it also combines a nest into one single loop but only if the stride is constant. When this happens, we can treat a two-dimension array as a linear one, using an incrementing pointer to access each one of the array positions.

Many other transformations exist [Wol95] like **Loop Tiling** that partitions a bi-dimensional iteration space into tiles to improve cache locality or **Loop Fission** that creates several loops from an original one, splitting the body statements through these new loops. One other loop transformation is **Loop Unrolling**, which has been studied and used for decades, and is the focus of this work.

1.1 Loop Unrolling

Loop Unrolling is a loop transformation that changes the loop structure. It replicates the loop body exposing successive iterations and adjusts the step accordingly. Because several iterations are grouped together the total number of iterations will decrease. The number of times the loop body is replicated is equal to the factor that the number of iterations decreases by and is called unroll factor. The step must also increase by this factor so that no iterations are executed more than once.

Figure 1.2 shows an example of this transformation, in this case with a loop unrolled twice. This loop calculates the sum of squares of a 512 element vector. Because the step of the transformed loop is 2 and the iteration limits remain the same, the number of iterations was reduced by half. Nevertheless, the number of statements inside the loop body also increased by 2, which means that even though the loop executes less iterations, the same logical operations are performed (further compiler optimizations might change the actual number of instructions). Because the execution order still preserves the original dependencies, Loop Unrolling is always legal.

The main advantage of this transformation is that it can immediately reduce the overhead associated with the loop control structure. But there are other advantages. On specific architectures it is possible to make use of Instruction Level Parallelism and Out of Order Execution. Loop Unrolling enables this as it exposes more iterations on the loop body, which in turn exposes more

```c
for ( i=0 ; i<512 ; i++ )
{
  prod = x[i] * x[i];
  sum += prod;
}
```

(a) Before

```c
for ( i=0 ; i<512 ; i+=2 )
{
  prod = x[i] * x[i];
  sum += prod;

  prod = x[i+1] * x[i+1];
  sum += prod;
}
```

(b) After

Figure 1.2: An example of Loop Unrolling with an unroll factor of 2. This loop performs the sum of the squares of a 512 element vector.

instructions. It is also possible, through the use of Loop Unrolling, to reuse data shared on different iterations, as they can now be closely together.

As it is expected, any source code transformation also carries some disadvantages. There are two main problems with Loop Unrolling and both arise from an increase on the source code size. First the growing number of instructions can lead to additional instruction cache misses. Second, there is a memory problem. Larger source files will be compiled to larger executable binaries. This can pose a problem for embedded systems, where memory is almost always a restraint.

Figure 1.3 presents the results of unrolling a loop that performs the sum of the squares of a 512 element vector (code presented in Figure 1.2). This example shows how much of an impact Loop Unrolling can have in the performance of the program. The figure presents the cycle count improvement for unroll factors ranging from 2 to 512 (which is a full unroll). This figure also shows how difficult it can be to predict the optimal unroll factor for a given loop as there are a lot of variations in performance and some pronounced spikes.

1.2 Problem Definition

As shown before, source code transformations, and especially loop transformations, can play an important role when attempting to improve the performance of applications. Sometimes, because of architecture, language or knowledge restrictions, these transformations can not be applied automatically. When this happens, the developer needs to take control and apply the transformations manually, a process that can be tedious and error-prone.

There is a need for a tool that can help developers make confident decisions about when and how to make these transformations. A tool that can help the developer feel safe when transforming the code because he knows it will lead to a performance improvement. Given a loop whose execution time we aim to reduce, we want to know if Loop Unrolling will be beneficial and, if so, what unroll factor should be used. For this, we need a tool that can analyze a loop and make a suggestion on whether we should apply this transformation, while also providing a suitable unroll factor if Loop Unrolling is suggested.

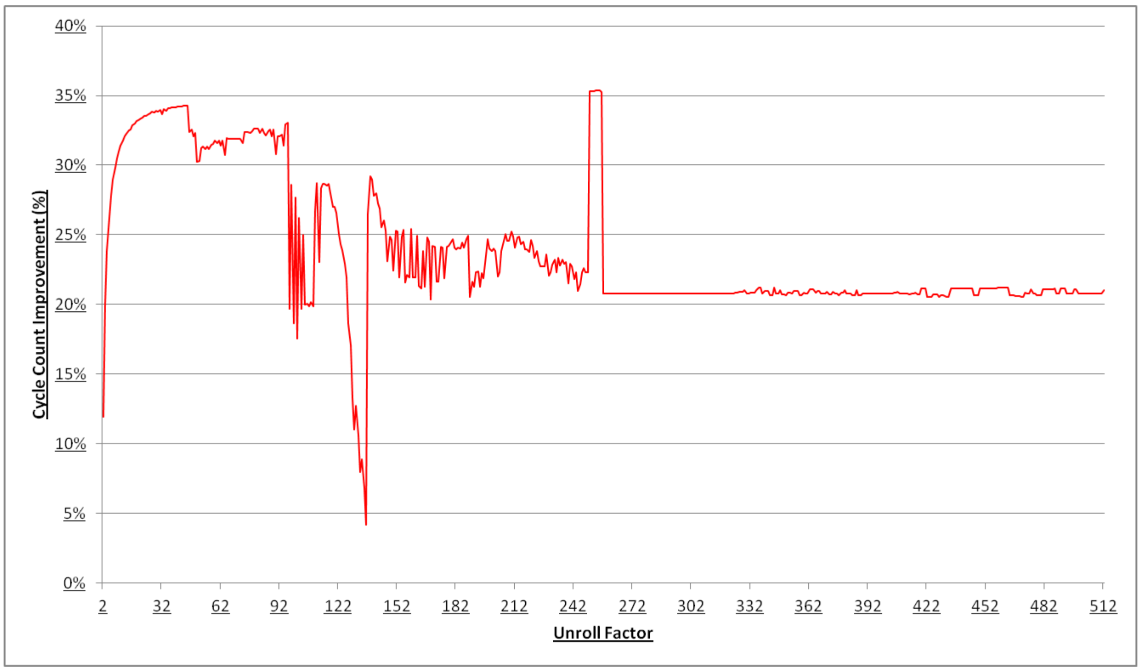


Figure 1.3: The results of applying Loop Unrolling to the loop presented in Figure 1.2. All the possible unroll factors were tested (from 2 to 512). These tests were done using a PowerPC 604 processor simulator.

1.3 Objectives

The main objective of this dissertation is to develop an approach that, using a set of heuristics, is capable of analyzing C source code and suggest loops that are good Loop Unrolling candidates. This approach will extract all the relevant information of candidate loops (innermost *FOR* loops) and create models to represent them. These models will then be evaluated using the proposed heuristics. After this, the loop will be considered suitable or unsuitable for Loop Unrolling. If a loop is considered suitable, we also need to suggest an appropriate unroll factor. This approach will tackle the problem presented earlier, helping the developer finding the best loops to unroll but without transforming them, giving him full control.

The choice of a source code transformation is obvious as it is the level that is closest to the developer and therefore is easier for him to understand, evaluate and apply. The choice of a loop transformation is also straightforward. As was mentioned before, optimizing loops will likely lead to a large increase in performance of the overall program. Out of all loop transformations, Loop Unrolling was chosen as it is very well known, studied and documented and has been used for several decades now. It is a transformation that is able to improve the performance of many loops, is easy to apply and has the advantage of always being a legal transformation.

It is also a goal of this dissertation to implement a prototype that can be used to validate and evaluate this approach. The prototype will make use of a compiler infrastructure as its front-end in order to get the input source file and generate an abstract representation of the source code. Another part of the prototype has the task of gathering all the information about candidate loops and creating models to represent them. These models can then be fed to an evaluation engine, also part of the prototype, that has an instance of the heuristics and evaluates each loop. The instance of the heuristics will target a specific architecture, considered meaningful in the scope of embedded applications development.

A secondary goal is that the heuristics to be developed can be used to evaluate loops as Loop Unrolling candidates for different architectures. To do this, the heuristics will focus on source level characteristics, which, as said before, is the target selected level.

1.4 Organization

The remainder of this document is organized as follows. Chapter 2 presents an overview of previous work that tries to somehow predict the impact of a transformation but also work focused on Loop Unrolling. In Chapter 3 it is possible to see the approach that is proposed in this thesis as a way to solve the current problem. Chapter 3 goes to fine detail and explains every part of the process. Next, Chapter 4 presents a prototype developed by creating an instance of the proposed approach, targeting a specific architecture. After that, Chapter 5 shows the experimental results of the evaluations conducted in order to validate and evaluate the prototype and therefore the approach. Chapter 6 draws the main conclusions and provides possible directions for future work. Finally, Appendix A presents the source code of the loops used as benchmarks in the evaluation experiments.

Chapter 2

Related Work

Some code optimizations can, on certain instances, cause some performance degradation. The main difficulty of applying code transformations is that it is hard to predict when they will be beneficial and when they will be harmful. Several studies have already tackled the problem of trying to predict the impact of these transformations [DC11], but currently we still do not have a systematic way of evaluating how a set of transformations will behave on a certain context. Predicting this impact is difficult as it is almost always architecture and/or machine dependent and can be affected by a large number of variables. Another factor is that it is necessary to take into account that applying any code transformation can have an impact on subsequent transformations [PCC09]. Often , these restrictive factors lead to the use of different approaches, such as iterative compilation [TVVA03]. The need for an uniform evaluation method of the impact of a transformation arises. This chapter presents several research projects that aim to solve this problem and explains how they addressed it.

This chapter is organized as follows. First, Section 2.1 presents three approaches that use analytical models. Section 2.2 presents two techniques that perform a solution-space exploration and Section 2.3 presents two machine learning approaches. Next, Section 2.4 presents several studies focused on Loop Unrolling. Finally, Section 2.5 presents and overview of all the approaches.

2.1 Analytical Models

This section presents three different approaches that use analytical models to predict the impact of several loop transformations.

2.1.1 Framework to Predict the Impact of Optimizations

Zhao et al. propose in [ZCS03] a framework that allows to predict, for an embedded processor, the impact of an optimization taking into account the available resources and the source code. This framework uses three types of models: code, resources and optimizations. This modular structure creates a high abstraction level and allows for great flexibility. To test the impact of a certain optimization on different resources, for example cache performance and energy consumption, only the resource model needs to be changed. It is also possible to combine several optimization models in order to predict the impact of applying a set of optimizations.

The code to be tested is converted to the code model. This model, alongside with the optimization parameters, is passed to the optimization model, which generates a new code model (representing the transformed code). To predict the impact of the optimization both the original code model and the new code model are tested by the resource model. The difference between the results of the two is the impact prediction.

For this work, it was created an instance of the framework whose goal is to predict the impact of the optimizations on the cache performance. Several loop transformations were also modelled, including Loop Unrolling and Loop Tiling. An already existent cache model was used. The results are positive as the framework was able to predict the impact of a transformation on cache performance with an average accuracy of 97%. For this particular instance, the framework managed to: apply transformations selectively; choose the best transformation when several are available; predict the impact of combining several transformations.

2.1.2 Combining Loop Transformations to Improve Cache Locality

In [MCT96], McKinley et al. present a cost model that can be used to decide the best loop organization inside a loop nest. This approach tries to improve data locality and, as such, the cost is greater the smaller the cache usage. A model was developed that is able to find the loop structure that makes fewer main memory accesses. In order to achieve this structure they use a composite transformation that uses Loop Permutation, Fusion, Distribution and Reversal.

This approach creates reference groups and calculates their cache reuse. This is used to calculate the reuse of each loop and its cost as if it was in the innermost position. After all loops are tested, the algorithm used tries to move the loop with lower cost to the innermost position, the loop with the second lower cost to the second innermost position and so on. The outermost loop would have the highest cost. When this optimal permutation (called *memory order*) is not legal, the algorithm tries to move the loop with the second lowest cost to the innermost position. If this fails, the third one is tested and so on. The other two transformations are mainly used as a way to remove dependencies that would prevent Loop Permutation from being used.

The model described in this work was implemented as part of Memoria, a source-to-source cache optimizer for Fortran applications. In the approach that was followed, Loop Permutation is the most significant transformation, with Fusion and Distribution also showing as capable of improving locality. While Loop Reversal was not capable of improving locality by itself, the results show that it is still an important transformation as its application can remove dependencies and allow for other transformations to be used.

2.1.3 Using Loop Transformations to Increase Parallelism of K-loops

In her Ph.D. dissertation, Dragomir [Dra11] presents an investigation about loop transformations for K-loops in the context of reconfigurable architectures. The architecture considered has a Field Programmable Gate Array (FPGA) and a General Purpose Processor (GPP). In this work, a K-loop is a loop that contains a kernel that will be executed on the reconfigurable hardware. The idea behind the application of the loop transformations is too extract as much parallelism as possible. This can be done by executing several instances of the kernel on the FPGA or by executing the software and hardware parts of the K-loop simultaneously. The problem with this is that a balance must be found between the performance and the area of the FPGA, which is a limited resource.

Several transformations were used in this dissertation. Loop Unrolling is used to expose parallelism and to have several instances of the kernel executing on the FPGA at the same time. Loop Shifting and K-pipelining are mainly used as a way of eliminating intra-iteration dependencies between the software and hardware parts of the K-loop. This allows for the operations of the loop to be concurrently executed on the FPGA and GPP. Finally, Loop Distribution is also used. When there is a loop with a large number of kernels, this transformation can be used to divide it into smaller loops allowing kernels to be analyzed individually.

The area available on the FPGA (90% of the total area to avoid routing problems) is used to set an upper bound to the unroll factor. Another bound is calculated using the time needed for the kernel to complete its memory accesses. This happens when the execution time of a kernel instance is completely overlapped by the time needed for the loads/stores of another instance. Yet another bound can be used. This is the relation between the performance gain obtained by increasing the unroll factor by one and the area needed to accommodate another kernel instance. It is defined by the developer and it indicates if it is still worth to increase the unroll factor considering the FPGA area needed. To decide which transformations should be applied, an algorithm that uses some heuristics is used. Relations of mutual exclusivity between transformations are also considered. After deciding the transformations to apply the unroll factor is chosen. The approach followed uses mathematical models for each of the possible transformation combinations and calculates the speedup and necessary FPGA area as a function of the unroll factor. These models need profiling information such as execution and memory access times. Now, using the upper bounds calculated before, it is possible to chose the best unroll factor.

The experimental results show that Loop Unrolling has a greater impact with faster software part execution times. When the execution time of the software part is much larger than the execution time of the hardware kernel, the impact of this transformation is negligible. It is also possible to conclude that it is always beneficial to apply Loop Shifting. When used in conjunction with Loop Unrolling, K-pipelining proves to be more useful than Loop Shifting. The experimental results also evidence that using Loop Distribution on large K-loops followed by the other transformations is better than just applying the other transformations as it allows for different kernels to use different unroll factors.

2.2 Solution-Space Exploration

This section presents two approaches that perform solution-space exploration in order to estimate the execution time after applying loop transformations.

2.2.1 Estimating Execution Time of Fully Unrolled Loops

In [CD04], Cardoso and Diniz present a model to predict the impact of full Loop Unrolling in the execution time and number of resources for Reconfigurable Processing Units (RPUs). Each loop is modelled using characteristics such as the number of iterations, the number of cycles in the Critical Path Length (CPL), the number of resources needed and also a status identifier (the loop can be pipelined, unrolled or in its original state). To calculate the execution cost of a loop three equations are used, one for each possible loop state. The arguments of these functions are extracted from the model mentioned above.

Each array reference on the code is represented by a Reference Vector (RV) of N elements, where N is the maximum number of loops plus one. The first element of this vector is the code root and the remaining represent the various levels of the loop nest, from the outermost to the innermost. Each position has a number that represents the number of references to that array on that level of the nest. When a loop is unrolled this number is updated, multiplying the number of iterations (of the loop being fully unrolled) by the number of references on the RV, starting at the level itself and proceeding to the left, until the outer loop.

The RVs are used to automatically generate all possible results of full Loop Unrolling for every loop, making it possible to estimate the execution time and the resources needed. The execution time is estimated using one of the three equations mentioned earlier and the resources needed follow a simple formula: multiply the number of necessary resources of the loop body by the number of unrolled iterations of that loop.

Despite the good results the model shows some discrepancies between the predicted execution times and the observed times, which can be explained by the fact that the model does not account for cases where a perfect pipeline balance is not reached. Their approach has the advantage of not needing a perfect loop nest and it provides a way to explore the solution-space without having to apply the transformation.

2.2.2 Estimating Execution Time of Singly Nested Loop Nests

Wolf et al. present a model [WMC96] that is able to estimate the execution time of transformed code taking into account cache misses, pipelining, register pressure and loop overhead. It enumerates the search space of applying all possible transformation combinations and chooses the combination with the best estimated performance. Search space pruning allows to do this efficiently.

The approach is based on the concept of Singly Nested Loop Nest (SNLN), which is a perfectly nested loop or an imperfect one where the imperfect code does not have any loop or flux control. This approach uses a three-phased algorithm.

The first phase uses Loop Fusion and Distribution to create deep SNLNs. For a given outer loop it recursively creates all SNLNs for its children and then uses Fusion and Distribution to combine the SNLNs with their parents. The second phase transforms the resulting SNLNs. For each one there is an algorithm that finds the best values for the unroll factor and tilling size as well as the best permutation. The third phase deals with loops that can not be register allocated, using Loop Distribution to create smaller loops.

To evaluate each combination this approach uses two models, one for the processor and one for the cache. The processor model tries to predict how the scheduling will be done and ignores cache effects. It accounts for three types of restrictions: resources, dependency caused latency and registers. The cache model is used to determine what tile size to use and to estimate the cache overhead. The processor model gives an estimate of the execution time and the cache model uses it to see how cache misses affect that execution time.

In order to limit the space explored when trying to find the solution, several restrictions are used. The product of the unroll factors used in a SNLN cannot be greater than 16. The search is also limited to nests with a maximum depth of 8. Finally, when choosing tile sizes, it is mandatory that the size is the same for every loop in the nest with the exception of the innermost loop.

The algorithm presented is part of a production compiler, MIPSPro 7 [mip]. It is capable of predicting the impact of several transformations by estimating their execution time. While at first, the algorithm might seem inefficient, the results of the experimental tests show that it leads to performance gains and that it does not take a long time to run. In the worst case scenario, the time needed for the algorithm accounts for 22% of compilation time.

2.3 Machine Learning Approaches

This section presents two machine learning approaches. The first is used to find a good set of heuristics and the second is used to find the best area to explore when performing iterative compilation.

2.3.1 Creating Architecture-Specific Compiler Heuristics

Monsifrot et al. present in [MBQ02] a new method that uses machine learning to find the best heuristics for Loop Unrolling on a specific architecture. The idea is that a heuristic is only meaningful if it is developed with knowledge of the target architecture. The authors consider that the parameters that can affect Loop Unrolling are mainly loop dependent as this transformation mostly influences code generation and instruction scheduling.

It was created an abstraction that represents a loop and it has 6 characteristics believed to affect performance. These were picked through cross validation and fall in one of five groups: memory accesses, number of arithmetic operations, loop body size, control instructions and iteration count.

Each example on the learning set, represented using the abstraction mentioned before, is tested twice. First on its original form and then after being transformed. The execution times can then be compared and it is possible to see if Unrolling is beneficial. After being tested, an example can be classified as: good, bad, non significant and equal. Non significant loops are discarded from the set.

The loops are grouped in equivalence classes. Loops with the same abstraction belong to the same class. The next step is to determine, for each class, if it is a positive or a negative example. Every loop that is considered bad can be counted several times, depending on how much the performance loss was. A class is considered a positive example if the number of good loops is greater than the number of bad loops.

The approach followed in this work was to create decision trees as they can be easily generated from characteristic vectors. The trees were created using OC1 [MKS94] technology and improved using a technique called boosting [FS95, FS99].

A set of tests was ran under two different architectures. The execution times of reference versions (compiled without Loop Unrolling) were compared to the execution times of code compiled after Loop Unrolling was applied (using the decision trees to decide when and how). In one of the architectures the average execution time was 93.8% of the reference time and on the other it was 96%. The final experiment swapped the decision trees. This resulted in a performance loss, proving that the heuristics were in fact architecture dependent.

2.3.2 Finding the Best Exploration Area to Improve Iterative Compilation

Agakov et al. [ABC+06] present a new methodology that aims to reduce the time necessary to the execution evaluation when applying iterative compilation. The authors use machine learning to find the most beneficial areas of the solution-space to explore. This approach is independent of the search algorithm, the solution-space and even the compiler. For each program an appropriate model is selected that indicates where the exploration should focus.

A total of eleven transformations were used and every transformation sequence with a length up to five was evaluated. Two different search methods were used, a random one and a genetic algorithm.

Two models were developed that indicate, for a certain type of program, which transformation sequences lead to the best performance. The first model, Independent Identically Distributed (IID), considers all transformations to be independent. The second model, a Markov model, considers that the impact of a transformation depends on the transformations that were previously applied. For each test in the learning set both models were created.

The models are used by search methods. When using the random search method, instead of having a uniform probability of each transformation being selected, each model favours some transformations. In the case of the genetic algorithm, the models are used to define the initial population.

Each program is represented by a vector with five characteristics. For each test used in the learning process, a vector and the two models were created. When a new program is to be tested, it is only necessary to convert it to this vector and, using a nearest neighbour approach, select the test case that resembles it the most. The models used for the new program are the models of the test case which is considered the nearest neighbour.

The results of the experimental tests are positive. In some cases it is possible to obtain a performance gain that the initial method could only achieve with ten times more evaluations. However, the approach has a disadvantage, which is that every new program that is tested is not included in the knowledge base.

2.4 Loop Unrolling

One of the reasons for choosing Loop Unrolling is that it is a widely know and studied loop transformation.

As mentioned previously, one of the main disadvantages of Loop Unrolling is the increasing number of instructions inside the loop body. This can lead to an instruction cache overflow. Dongarra and Hinds present an early study [DH79] of how Loop Unrolling can effectively improve the performance of Fortran loops. On this work they notice the existence of a constraint to the application of this transformation, which is the possibility of overflowing the *"instruction stack"*. Weiss and Smith [WS87] study two compilation techniques, Unrolling and Software Pipelining, for pipelined supercomputers. When considering Loop Unrolling they investigate how the instruction buffer size affects its efficiency. They show that even though this is a software transformation, the hardware supporting it plays an important role.

In order to facilitate analysis, most loops that are considered for Loop Unrolling are loops where the iteration count is known at compile-time. In [DJ01], Davidson and Jinturkar, propose a more aggressive approach in order to make use of most loops. The authors defend that in most loops it is not possible to know the iteration bounds at compile-time and that if those loops are discarded a large number of optimization chances will be lost. When the number of iterations is only found at execution-time, it is not possible to chose an unroll factor that is a divisor of the number of iterations. In this case, a special structure (either an epilogue or a prologue) is needed to account for left over iterations. The authors believe that it is better to apply this transformation automatically and in a later stage of the compilation process as more information is available. This work presents an algorithm that is capable of dealing with loops whose iteration count is only known at execution-time. The experimental results show that their aggressive approach is capable of improvements as high as 50% with an average performance improvement of 10%.

One of the hardest tasks when deciding whether to apply Loop Unrolling is to decide which unroll factor to use. Several studies [KKF97, DMBW08, Sar00, SA05] were dedicated to this, targeting different applications and systems. These range from trying to find suitable unroll factors for perfect loop nests considering instruction caches and Instruction Level Parallelism, to using Machine Learning approaches as a way of predicting unroll factors.

Some authors studied how Loop Unrolling relates to other transformations [DJ01, WS90]. This is important because Loop Unrolling is know as a facilitator for other source code transformations. Because it exposes more code, Unrolling facilitates the analysis and application of other transformations. On the other hand, it might also prevent their application. Imagine we wanted to inline a function call inside a loop. After applying Unrolling, Function Inlining might no longer be possible, as now, there is a chance of creating a hotspot that can lead to problems like increasing instruction cache misses.

2.5 Overview

Zhao et al. [ZCS03] propose a framework to predict the impact of applying a transformation, in a code context for a given resource. In their work it is presented an instance of the framework that targets cache performance.

McKinley et al. [MCT96] present an algorithm that manages, using a small set of transformations, to find the loop nest structure that leads to the best cache performance. The algorithm is implemented as part of a source-to-source optimizer, Memoria.

Dragomir [Dra11] presents an intensive study about how several transformations affect performance in the context of reconfigurable architectures. Several mathematical models are proposed to find the combination of transformations that manages to explore maximum parallelism while accounting for memory and area restrictions on the reconfigurable devices.

Cardoso and Diniz propose a methodology [CD04] that predicts the impact of fully unrolling a loop by means of a solution-space exploration. It is capable of predicting execution times and the number of resources necessary. Their work presents an efficient method of exploring the solution-space.

Wolf et al. [WMC96] present an algorithm capable of predicting the execution time of a program for a specific architecture. It needs a large amount of architecture information as it accounts for cache, registers, loop overhead and software pipelining. The algorithm is implemented as part of a production compiler, MIPSPro 7.

Monsifrot et al. [MBQ02] use Loop Unrolling to demonstrate that machine learning can be used to find the best heuristics for a given architecture. At the end, it is proved that the heuristics created are, in fact, architecture targeted.

Agakov et al. [ABC$^+$06] show how machine learning techniques can be used to limit the search space, increasing the speed of iterative optimizations. This methodology is able to indicate, for a given program, the areas of the solution-space where the search should be focused. The main advantage of this methodology is that it is independent of the solution-space, the search algorithm and the compiler/optimizer used.

There are several different approaches. Some of them use analytical models, others explore the solution-space generated by the possible application of the various transformations and some others use machine learning techniques. Different methodologies are also used. For some of the approaches the architecture plays a critical role, while others manage to completely abstract themselves from it. The results are different too. Some of the approaches try to predict the impact a transformation will have on performance and others try to find the best set of heuristics that allows to decided when and how to apply a transformation. Whatever the approach is, the ultimate goal is always the same: given a certain context, find the transformation (or combination of transformations) that will lead to the best performance improvement. Table 2.1 presents a summary of all the approaches that were presented in this chapter.

2.6 Summary

This chapter presented several research efforts that have in common the fact that, for a given source code, they intend to estimate the impact of a source code transformation. Three main approaches were presented but even in each one of them there are differences in the goals and on the results obtained.

Loop Unrolling was chosen as it is one of the most used loop transformations. It has been used for decades it is vastly studied. It is also a very simple transformation to understand and apply and has one big advantage in the fact that its applicability is always legal.

Although the many presented approaches proposed methods of predicting the impact of loop transformations, an universal approach to suggest the application of Loop Unrolling (and a suitable unroll factor) is still dependent on the developer.

Table 2.1: A summary of the approaches presented in this chapter.

Reference	Transformations	Input	Output	Application	Implementation
[ZCS03]	Loop Unrolling, Interchange, Tiling, Reversal, Fusion and Fission	Source code, resource model and optimization parameters	Prediction of the impact in cache misses	Embedded systems	n/a
[MCT96]	Interchange, Reversal, Fusion and Fission	Source code and cache line size	Best loop nest organization for cache	n/a	Memoria
[Dra11]	Loop Unrolling, Shifting, K-pipelining and Fission	Source code, FPGA information, profiling information and f factor	Best unroll factor, best loop nest organization and best sequence for SW/HW parallelism	Reconfigurable architectures	n/a
[CD04]	Full Loop Unrolling	Source code	Execution time estimate and resource needs estimate	Reconfigurable architectures	n/a
[WMC96]	Unroll-and-Jam, Fusion, Fission, Interchange and Tiling	Source code and architecture information	Execution time estimate (in cycles)	n/a	MIPSPro 7
[MBQ02]	Loop Unrolling	Source code and learning set	Best heuristics for Unrolling for a certain architecture	n/a	n/a
[ABC+06]	Loop Unrolling, Commmon Sub-expression Elimination and 9 more	Source code and learning set	Probability distribution indicating the best transformations	Iterative optimization	n/a

Chapter 3

Proposed Approach

This chapter describes how a loop can be evaluated in order to find whether it is suitable for Loop Unrolling. Initially, the source code is translated to a model that represents the loop that we want to analyze. After this, the loop is tested against a set of heuristics and if it is suggested for Unrolling, a suitable unroll factor is chosen. This chapter describes this process in detail and specifies the loop model used, the heuristics and the metrics used to chose the unroll factor.

This chapter is organized as follows. Section 3.1 describes the flow of the evaluation process, from the parsing of the source file to the suggestion of the unroll factor. Section 3.2 presents the loop model that is used to represent the loops. Next, in Section 3.3, we present set of rules used to discard loops that may not be suitable for Loop Unrolling. The heuristics used to score the loop are presented in Section 3.4. Finally, Section 3.5 explains how we can propose good Loop Unrolling factors.

3.1 Evaluation Process

The evaluation has four main steps. First we need to take the source code and create loop models for every loop that we want to analyze, i.e, innermost *FOR* loops whose iteration count is known at compile-time. The next step, tests the loop against a set of three rules. These rules are used as a way of filtering loops that we know beforehand, are unsuitable for Loop Unrolling. The third step uses a set of heuristics to evaluate all the loops that passed the second step. After being evaluated, a loop will have a final score, which is the sum of the values given by each one of the heuristics. If this value is below a threshold, then the loop is not considered a good candidate for Unrolling and it can be discarded. If however, the loop has a positive score at the end of the evaluation it will move on to the fourth and final step. This step is the choice of the unroll factor. The choice is based on some metrics that are machine dependant and on characteristics of the code. During the third step, the loop model was flagged when certain characteristics were present, e.g., if it has a small iteration count. These flags are used when deciding what the unroll factor should be.

Continuing the small iteration count example, when a loop has that flag it will be considered for full Loop Unrolling.

Initially we need to convert the C source code we want to analyze to our loop model (presented in Section 3.2), so that we can easily test for the presence of certain characteristics. In order to do this we can take a compiler front-end and use it to build an Abstract Syntax Tree (AST). We can then navigate that AST and collect all the needed information. First we look for innermost loops and then, for each one of them, we extract the characteristics that matter for our analysis. Now we have a loop model for every possible loop that we would want to analyze and can proceed to the next step.

The acceptance rules (presented in Section 3.3) check for the existence of characteristics that make a loop unsuitable for transformation. There are only three acceptance rules and every one of them is easy to test. All of the rules have one main goal which is to make sure that it is possible to know, at compile-time, how many iterations the loop will execute. If a loop breaks even one of this rules then it will be immediately considered a bad candidate for Loop Unrolling and will be discarded. On the other hand, if the loop is in compliance with these rules then we can begin its evaluation using the proposed heuristics. When we reach this point the loop has a score of zero. This score will change as it is tested by the several heuristics.

The third step will evaluate the loop considering a set of heuristics (presented in Section 3.4). First we look for characteristics that are considered good and then for characteristics that might make the loop inadequate for Loop Unrolling. These heuristics will change the loop score, that initially has a value of zero. After this evaluation, and depending on the final loop score, we may or may not suggest the loop for transformation. If it is suggested then we need to choose a fitting unroll factor.

Figure 3.1 presents an activity diagram of the evaluation process. We start by using a tool to convert the source file into an AST and then generating our loop model using that AST. With our loop model we can test the acceptance rules, evaluate the loop and suggest a good unroll factor.

The process described in this section could be used iteratively. That is, if we had a source-to-source transformation tool that would Unroll the loop based on our suggestion, we could pass the transformed source file back into the evaluator. This would allow for some exploration of the solution space if we were to change the values used by the heuristics after each run. With this approach it would also be possible to consider loop nests (only to a certain extent), something that is not considered currently. If we had a nested loop and the innermost loop was fully unrolled, the outer loop could then be evaluated too. This is currently possible but the Loop Unrolling transformation has to be done externally and the resulting source file must be passed ot the evaluator manually.

3.2 Loop Model

This section presents the loop model that is used to represent a loop. This model is created after parsing the source code that is being analyzed. The need for this model comes from the fact that

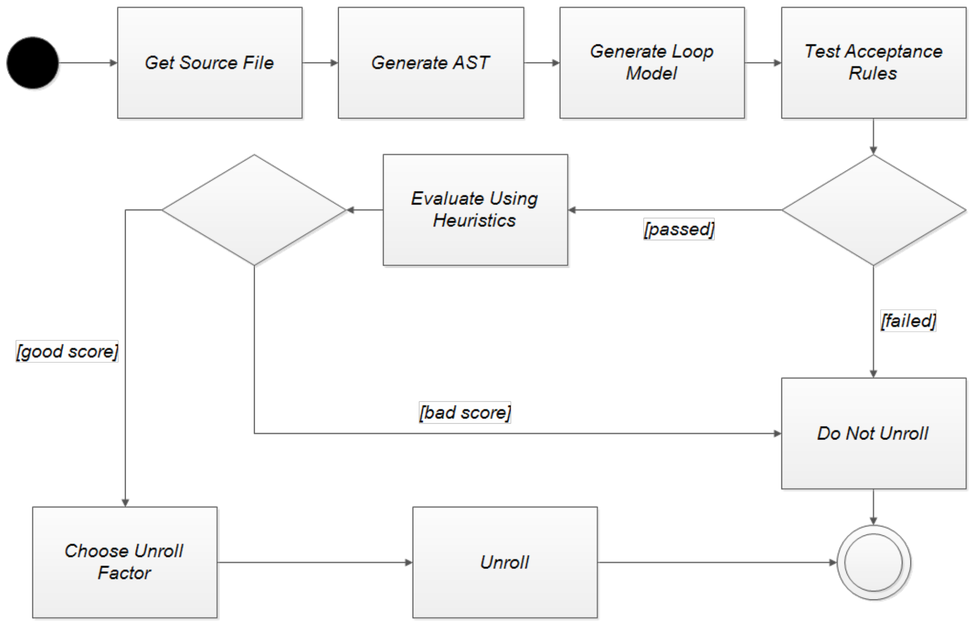


Figure 3.1: An activity diagram for the evaluation process described in this section.

to evaluate a loop we need to know information about it and this is a good way to represent said information and retrieve it quickly. This is purely a code model, i.e., it only represents information present on the source code. This model can be built, for example, by extracting information from an Abstract Syntax Tree created by any compiler.

This model is created to gather all the possibly needed information about a loop in a way that is both easy and fast to query. All the information is divided into three parts. This makes the process of evaluating the loop a lot easier. The information about the loop limits and iteration count is on the **Loop Header**. The statements that make up the body are stored on the **Loop Body** and the **Arrays** have information about the arrays that are accessed inside the loop body. Figure 3.2 presents a simple class diagram (no methods are displayed) that helps understand the overall structure of the model.

The Loop Header contains information about the number of iterations the loop executes. This is of crucial importance for a transformation such as Loop Unrolling. The Loop Header also contains the lower and upper bound limits, the step and the induction variable of the loop.

The Loop Body has the statements of the loop. These statements are used by most of the heuristics that will be presented next. Using these statements it is possible to calculate the number of assembly instructions that will be generated by the compiler, the smallest reuse distance and the loop body execution time relation, all of which will be explained ahead, in the heuristics section (see Section 3.4 on page 21).

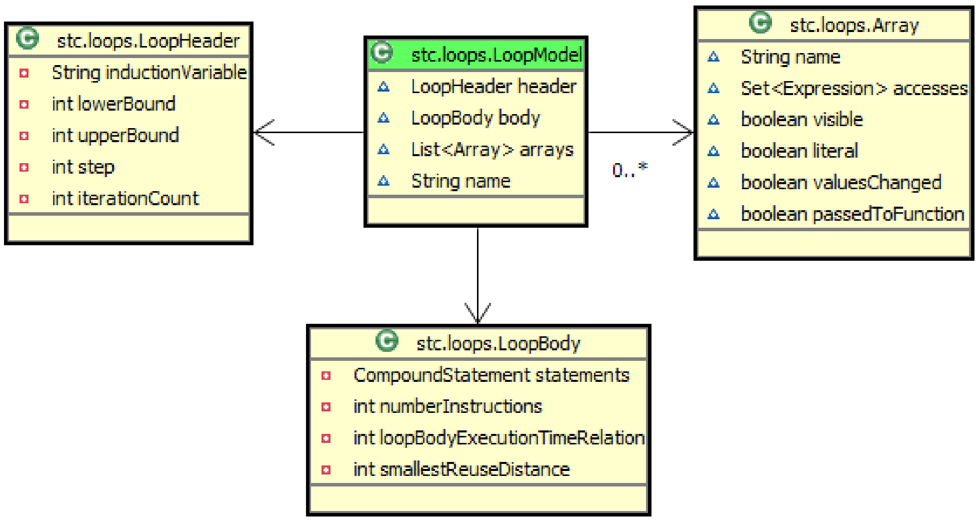


Figure 3.2: Class diagram for the Loop Model.

A loop can be modelled using a header, a body and a name, which is used as an identifier. Nevertheless, in order to make a better analysis of the loop, we also need information about the arrays that it iterates over, so each loop has a list of **Array** objects. Each one of these objects has all the needed information about an array that is being accessed inside the loop body. This allows us to test for characteristics such as data reuse. We need a name to identify each array. Then we need to have a set of all the accesses that are made to this array. For this, we store a set with the subscripts of the array accesses. This is useful to check, for example, which variables are used to index an array position. Finally, we want to know other characteristics about the array like if its declaration is visible to our loop and if its initialization uses only literals. We also want to know if the array values were changed before it reached our loop or if the array was passed to a function (were its values might have been changed). All this information is used by an heuristic that tries to understand if it possible, by applying full Loop Unrolling, to replace the array accesses with the array values (this is explained in detail in Section 3.4 on page 21).

3.3 Acceptance Rules

Some loops have certain characteristics that make them less suitable to transform with Loop Unrolling. For example, it may not be possible to know, when we are analyzing the loop, how many iterations it is going to execute. In cases like this it is better not to transform the loop. This section contains a set of rules that are used to detect these loop characteristics. When a loop breaks one of these rules then it can be classified as a bad candidate and discarded immediately. The loops that respect these rules will go to the next step where they will be evaluated.

```
for ( i=1 ; i<n ; i++ )
{
  for ( k=0 ; k<i ; k++ )
  {
    w[i] += b[k][i] * w[(i-k)-1];
  }
}
```

Figure 3.3: Livermore Loop no. 6.

Acceptance Rule 1 The loop will terminate its execution only when all of its iterations are complete. The loop body may not contain any instructions that can either terminate the loop or skip iterations (*continue*, *break*, *goto*).

Acceptance Rule 2 The Upper Bound of the loop must remain unchanged throughout its execution. Even in the case where this value is only known at execution-time it must be a constant. No instructions inside the loop can change its value. While this value can not be change inside the loop, it might be changed outside (see Figure 3.3). This is not accounted for in our approach.

Acceptance Rule 3 While the loop is executing, the step must remain constant. No instructions inside the loop body are allowed to change the induction variable nor the variables used as step.

These three rules make it possible to know the number of iterations and apply Loop Unrolling. The number of iterations is given by:

$$\left\lfloor \frac{UpperBound - LowerBound}{Step} \right\rfloor$$

3.4 Heuristics

This section describes the heuristics used to classify a loop as either a good or a bad candidate for Loop Unrolling. This is the third step for suggesting Loop Unrolling as a transformation for a given loop. The first four heuristics are called **positive heuristics**. These represent good loop characteristics and because of this, their value is always positive or zero.

The last heuristic is a **negative heuristic** as it looks for a characteristic that tends to have a negative impact once Loop Unrolling is applied. Because of this, the values given when testing a loop will be either zero or negative.

3.4.1 Small Iteration Count

Loops with small iteration counts are considered good because it is not likely that Loop Unrolling will have a negative impact on the instruction cache performance. Even if the loop body has a large number of instructions, one can still fully unroll the loop without worrying about cache misses. Fully unrolling a loop completely removes the control structure used and exposes all the

```
1 for ( i=1 ; i<SIZE ; i++ )
2 {
3   x[i] = z[i] * ( y[i] - x[i-1] );
4 }
```

(a) Original loop

```
1 for ( i=1 ; i<SIZE ; i+=3 )
2 {
3   x[i]^a= z[i] * ( y[i] - x[i-1] );
4   x[i+1]^b= z[i+1] * ( y[i+1] - x[i]^a);
5   x[i+2] = z[i+2] * ( y[i+2] - x[i+1]^b);
6 }
```

(b) Loop unrolled three times

Figure 3.4: An example of a loop with a data reuse distance of 1. This is the fifth loop of the Livermore Loops benchmark. In (b), it is possible to see the two reused values, annotated with the *a* and *b* superscripts.

instructions. Not only can this improve performance because it removes the overhead associated with the control structure but it can also lead to further compiler optimizations.

This heuristic looks for a loop whose total number of iterations is small. Depending on how small the iteration count is, a different value will be added to the loop score. To do this we need to consult the *Loop Header*, get the number of iterations and return a suitable value. The score of this heuristic is higher for loops with smaller iteration counts. If the loop is considered to have a small iteration count, then it is flagged as such. This is done as a way to advice the code responsible for choosing the unroll factor to try full Loop Unrolling.

3.4.2 Data Reuse

Alongside with the overhead reduction, data reuse is one of the most important benefits of Loop Unrolling. This transformation exposes several iterations inside the loop body, making it possible to reuse data from several iterations on one single iteration. If there is reuse, we need to know what is the distance between the iterations that use the same data. Short distances are better and will be rewarded higher. Long distances mean that we need a bigger unroll factor in order to reuse the data and bigger unroll factors mean more problems with instruction cache misses. When the same unroll factor is used, a loop with a shorter reuse distance will be able to reuse more data than a loop with a longer reuse distance.

Let us consider the code shown on Figure 3.4 as an example. This is the code for the Livermore Loop no. 5, part of the Livermore Loops benchmark. The statement on line 3 of Figure 3.4a uses data that is written in previous iterations. When the loop is unrolled, these iterations are exposed and the data can be reused. Since there was a distance of one, unrolling the loop three times allows for two values to be reused. Lines 4 and 5 of Figure 3.4b show statements that reuse data from the previous statement (the reused values are annotated with the *a* and *b* superscripts).

The value of this heuristic depends on how long the reuse distance is, with shorter distances having higher scores. To test this heuristic we need to see what array accesses are inside the loop body and if they allow data to be reused. If this characteristic is found it not only marks the loop as a good candidate, but also sets a possible value for the unroll factor (reuse distance plus 1) that will be chosen on the next evaluation step.

3.4.3 Same Scope Array

Most loops iterate over arrays. If these arrays are declared on the same scope that the loop is, then it can be beneficial to fully unroll this loop. When such an array exists and its values are not reassigned (i.e., it is an array of constants) fully unrolling the loop can significantly improve performance. With the loop fully unrolled, it is possible, through constant propagation and constant folding, to resolve the array indexes and replace the array accesses with the constant values. While this is not a common occurrence it only needs a simple verification and might bring a very good performance improvement as not only it removes the loop control structure overhead but also removes unnecessary loads.

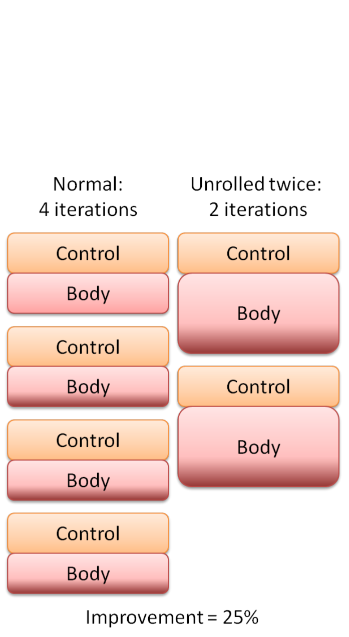
In order to test this heuristic we need to see what arrays are accessed inside the loop and if the subscripts of these accesses are only dependent on its induction variable. Furthermore, we need to make sure that the array we are considering is declared on the same scope and is an array of constants.

3.4.4 Loop Body Execution Time Relation

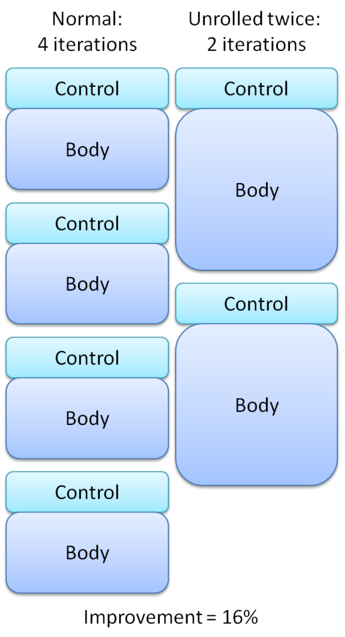
One of the main advantages of Loop Unrolling is that it greatly reduces the overhead associated with the control structure of the loop. The longer the execution time of the control structure in relation to the overall execution time of the loop, the greater the impact of this transformation. Considering that the time needed to execute the control is about the same for every loop (just an increment, a comparison and a jump), the only thing that can be tested is the execution time of the loop body. If the loop body takes a long time to execute than the transformation will not be as significant. We can consider the relation between the execution time of the body and the execution time of the control.

To illustrate this, see Figure 3.5. In 3.5a we see a loop with a relation of 1, meaning that the body and the control structure take the same time to execute. In this example the loop has only four iterations and we show the performance gain resulting from the control overhead reduction when unrolling the loop twice. After unrolling the loop, it only executes two iterations, so the control structure is only executed twice. This represents an improvement of 25%. On the other hand, in Figure 3.5b we have a loop whose relation is 2 (i.e., the body takes twice as long to execute when compared to the control). This loop also executes four iterations. After unrolling this loop (using a factor of 2 once again), we only achieve an improvement of 16%. This shows that the execution time of the loop body in relation to the execution time of the control structure should be as small as possible to achieve better performance gains.

This heuristic rewards loops where the loop body execution time is as small as possible when compared to the control structure. To test this, we assume that the control structure is composed of an increment, a comparison and a jump. Now we can use this information and the time needed to complete the loop body to calculate a relation between the execution time of the control and the body. First this test calculates the time needed to complete the loop body. The next step is to divide this time by the time needed by the control, which is the time needed to execute an increment, a



(a) A loop with a relation of 1



(b) A loop with a relation of 2

Figure 3.5: Comparing the performance gain resulting from Loop Unrolling on loops with different loop body execution times.

comparison and a jump. Based on the relation of these two values, we can return an appropriate score.

3.4.5 Number of Instructions

The number of instructions on the loop body should be small enough so that the instruction cache is not greatly affected by this transformation. Instruction cache misses are always going to be a problem but the bigger the number of instructions in the loop, the more cache misses will happen when applying Loop Unrolling. In this context, *"instructions"* refers to the assembly instructions on the loop body as generated by the compiler. Small numbers of instructions will grant a higher score and the score decreases as the number of instructions inside the loop body grows.

Table 3.1 presents a summary of all the heuristics that were presented. This table shows the name of the heuristic and the parameters it takes, i.e., the information needed to calculate its value. It can also be seen in this table what effect (that emerges from Loop Unrolling) the heuristics are looking for. Consider, for example, the *Number of Instructions* heuristic. It needs to know the number of instructions inside the loop body in order to account for the increasing instruction cache misses that can result from Loop Unrolling.

Table 3.1: A summary of all the heuristics described.

Heuristic	Parameter	Effect
Small Iteration Count	Number of iterations	Safer full unroll
Data Reuse	Array references	Data reuse
Same Scope Array	Array references and scope information	Array accesses replacement
Loop Body Execution Time Relation	Loop body statements	Control overhead reduction
Number of Instructions	Number of assembly instructions	Instruction cache thrashing

3.5 Unroll Factor

This section describes how the choice of the Unroll factor is made. After the loop is evaluated we need to choose an unroll factor that is capable of achieving a good performance gain. To do this we need to use the information that is present on the loop model described previously and limit the unroll factor using some metrics that allow to estimate when the performance will start to fall.

The strategy used to choose a good unroll factor is very simple, we try to choose the biggest unroll factor that will not lead to a performance loss. This assumes that all the loops that reach this stage are good candidates and that they will benefit from Loop Unrolling. There are three sequential and exclusive steps, i.e., when one of them is true the remaining are not tested. The algorithm described in the following paragraphs can be seen in Figure 3.6.

Initially, we check if the arrays that are being iterated on the loop are declared on the same scope and are constant (*SSA* in Figure 3.6, line 2). This was already tested by the heuristics and if this is true the loop is correctly flagged and we fully unroll the loop. While this might seem a bit drastic the performance increase that results from the replacement of the array accesses by the array values is great and can compensate most of the negative effects inherent to Loop Unrolling. However, if this is not true, we move on to the next test.

Now we need to calculate the maximum possible unroll factor (from now on *MaxFactor*) that will not cause problems with the instruction cache. To do this we use a metric, *MaxInsts* that represents the maximum number of assembly instructions that will not cause the instruction cache performance to deteriorate in way that is not possible to compensate with the benefits from Loop Unrolling. Right now, we assume that the only possible cause for performance degradation is the instruction cache. To calculate *MaxFactor* we do the following:

$$MaxFactor = \left\lfloor \frac{MaxInsts}{LoopBodyInsts} \right\rfloor,$$

where *LoopBodyInsts* is the number of assembly instructions on the loop body. Now that we have this upper limit we can proceed with the other two steps.

At this point we check if the loop has a small number of iterations (*SIC* in Figure 3.6, line 6). If it does and the number of iterations is smaller than *MaxFactor*, the loop can be fully unrolled. If the number of iterations, even thought it is considered small, is bigger than *MaxFactor* we need to check if it is still worth to fully unroll the loop. We calculate the difference between the number of iterations (the full unroll factor) and *MaxFactor* and we compare it to *MaxFull*. This metric has

```
function ChooseFactor($LoopBodyInsts, IterationCount, ReuseDistance$)
    if $SSA = true$ then
        return $FullFactor$
    end if
    $MaxFactor \leftarrow \left\lfloor \frac{MaxInsts}{LoopBodyInsts} \right\rfloor$
    if $SIC = true$ then
        if $IterationCount < MaxFactor$ then
            return $FullFactor$
        else
            $Difference = IterationCount - MaxFactor$
            if $Difference \leq MaxFull$ then
                return $FullFactor$
            end if
        end if
    end if
    if $DR = true$ then
        $ReuseFactor \leftarrow ReuseDistance + 1$
        if $ReuseFactor < MaxFactor$ then
            return $ReuseFactor$
        else
            $Difference = ReuseFactor - MaxFactor$
            if $Difference \leq MaxDistance$ then
                return $ReuseFactor$
            end if
        end if
    end if
    return $MaxFactor$
end function
```

Figure 3.6: Calculating a suitable unroll factor. This figure shows the algorithm used to choose a good unroll factor to the loop being analysed. It uses the following information about the loop: the number of instructions on the body, the iteration count and the smallest reuse distance.

the maximum difference between the iteration count and *MaxFactor* that will still allow to fully unroll the loop. If the difference is smaller than *MaxFull*, we fully unroll the loop. Otherwise, we proceed to the next test.

Now we see if the loop has the possibility of data reuse (*DR* in Figure 3.6, line 16). If it has, we need to see what is the reuse distance and how it compares to *MaxFactor*. In order to reuse data with Loop Unrolling we need an unroll factor that is at least the reuse distance plus one (from now on *ReuseFactor*). If *ReuseFactor* is smaller than *MaxFactor* we can safely use it and we are guaranteed to reuse a number of values equal to:

$$MaxFactor - ReuseFactor + 1$$

If *ReuseFactor* happens to be bigger, we need to see if it is still worth using it as the unroll factor. In a similar way to what we did on the previous test we calculate the difference between *ReuseFactor* and *MaxFactor* and compare it to another metric, *MaxDistance*. In an analogous way to *MaxFull*, this metric tells us if it is still worth using *ReuseFactor* as the unroll factor.

If all the previous tests failed we say that the best unroll factor is *MaxFactor*, the value that was calculated between the first and the second steps. This is what we try to do in a summarized way:

- Fully unroll if the array is constant and declared on the same scope.
- Try to fully unroll if the loop has a small iteration count.
- Try to reuse data if possible, using "reuse distance + 1" as the unroll factor.
- Return the maximum possible unroll factor if everything else fails.

3.6 Summary

This chapter presented an approach that can be used to evaluate a loop and make an informed suggestion of whether it should be unrolled or not.

In order to evaluate a loop there are four main steps. In the first step the source code is translated to a loop model that has all the relevant information about the loops that will be analyzed. Next, we need to test the loops against a set of acceptance rules. These rules will act as a filter, discarding loops on which it is not possible to know the number of iterations at compile-time. The third step is the evaluation of the loop. The loop starts with a score of zero and will be rewarded points as the heuristics find characteristics that make this loop suitable for Loop Unrolling. Finally, and if the loop has a final score above a threshold, the fourth and final step tries to find a fitting unroll factor.

The main advantages that arise from using Loop Unrolling are the control structure overhead reduction, the possibility of data reuse and the possibility of replacing array accesses with the array values. The control structure overhead reduction is the one advantage that is always present. By control structure we mean the instructions responsible for making sure that the induction variable is updated every iteration and that the loop is terminated. Because these instructions are executed on every single iteration and Loop unrolling reduces the number of iterations we can be sure that we can reduce the time spent on them.

On some loops, the same data is used in different iterations. Because Loop Unrolling exposes several iterations it is possible to reuse this data inside the same iteration after the transformation. This can reduce the number of loads and greatly improve performance. The number of iterations that sets the reuse apart is the reuse distance. The shorter the distance the better, as shorter distances need smaller unroll factors in order to enable data reuse.

Certain loops iterate over arrays of constants. If these arrays are declared on the same scope of the loop, it might be possible, after fully unrolling the loop, to replace the array accesses with its values. When a loop is fully unrolled it is possible, using constant propagation and constant folding, to resolve the array subscripts and know exactly what position is being referenced. When this happens and we know the array is constant we can just replace the accesses. These are the three main advantages of Loop Unrolling considered on this work and are the ones that the positive heuristics will look for. The loop score will increase if these characteristics are found.

Loop Unrolling also has disadvantages such as increasing the instruction cache misses. Because this can become a problem, there is also a negative heuristic that looks for a large number of instructions inside the loop body.

To choose a good unroll factor a simple strategy is used. We assume that whatever loop reaches this step will benefit from Loop Unrolling. Considering this we choose the biggest possible unroll factor that will not cause the instruction cache thrashing to overbalance the performance gains achieved through Loop Unrolling.

Chapter 4

Prototype

This chapter presents a prototype implementing the approach described previously (see Chapter 3, page 17). In this dissertation, the prototype uses heuristic values tuned to a target processor, the PowerPC 604, a member of the PowerPC architecture. This prototype uses Cetus as its front-end and extends it in order to gather all the necessary information for the loop model to be evaluated. After the evaluation, which is done by an evaluation engine that received the loop model, the suggestion is written as a comment on the Abstract Syntax Tree (AST) which is later dumped to a source file.

The organization of this chapter is as follows. Section 4.1 describes how the several parts of the prototype are used together. Section 4.2 presents Cetus, the tool used as a front-end in this prototype. Next, in Section 4.3 we include an explanation of the software that was developed and integrated in Cetus. Finally, Section 4.4 presents the values taken by the heuristics and the metrics used to chose an unroll factor for this particular instance of the proposed approach.

4.1 Prototype Process

The prototype receives a C source code file as its input. This file is parsed and it is converted to an AST. Both the source file parsing and the AST generation are done by **Cetus**. The next step is to extract all the relevant information from this AST and compile it on the previously explained loop model (see Section 3.2 on page 18). To extract this information from the AST we use the **Bridge**, which is the software responsible for navigating the AST and finding the code characteristics we are looking for.

When the loop model is ready we can use it to evaluate the loop. For this we use an instance of the **Evaluation Engine**. This engine instantiates the heuristics from a configuration file and then tests them using the information that is present on the loop model. The values of this heuristics are presented on a later section in this chapter. For every combination of heuristic configuration file and loop model, the engine will produce a score. This score is used to know whether the loop

```
/*
  Score: 38
  Unroll with factor: 3
*/
for (i=0; i<3; i ++ )
{
  array_p[i]++;
}
```

Figure 4.1: An example of how the suggestion is done as a comment on the source code.

should be suggested for Unrolling or not. In the case of this prototype, the threshold used is 0. So if the loop has a positive score it is considered a good candidate.

Finally, we can suggest an unroll factor that suits the loop. To do this we take into account the number of assembly instructions that the compiler will generate when compiling the loop body. The values used to make this suggestion are explained further ahead in this chapter. When we find a suitable unroll factor we annotate the loop with it. Figure 4.1 shows an example of this. Because Cetus is a source-to-source tool we can change the code through the Intermediate Representation (IR) tree to create that comment and then output the resulting code to another source code file.

Figure 4.2 presents an activity diagram that was shown previously to explain the activity flow used to evaluate a loop. In this figure, however, it is possible to see which part of the prototype does what. The activities marked with the letter *C* are done using Cetus. These include the first two activities, parsing the source file and building the AST, as well as the last two activities, annotating the source file with the correct suggestion. The activities marked with the letter *E* are performed by the Evaluation Engine. This engine will evaluate a loop according to the heuristic values and choose a good unroll factor. Marked with the letter *B*, we have the only activity that the Bridge is responsible for, that is gathering all the relevant information from the AST, compiling it inside an instance of the loop model and pass it to the evaluation engine.

4.2 Cetus

In order to parse the source code files and create some kind of IR a compiler front-end was needed. The chosen tool was Cetus [Uni11, DBM$^+$09], a compiler infrastructure used for source-to-source analysis and transformation. It is written in Java and can be extended using this same language. The only built-in parser (written using ANTLR) accepts C source code files although other front-ends, that would parse any source file and convert it to Cetus IR, could be added. The only language that is fully supported is ANSI C, which is not a problem considering the type of applications targeted.

After building the AST, Cetus provides us with the means necessary to navigate the tree (both depth-first and breadth-first) and retrieve all the information that we need from the code. We also could, although it is not in the scope of this dissertation, use Cetus to change the source code, by altering the IR and then outputting code to another file. This could be used, for example, to automatically apply Loop Unrolling.

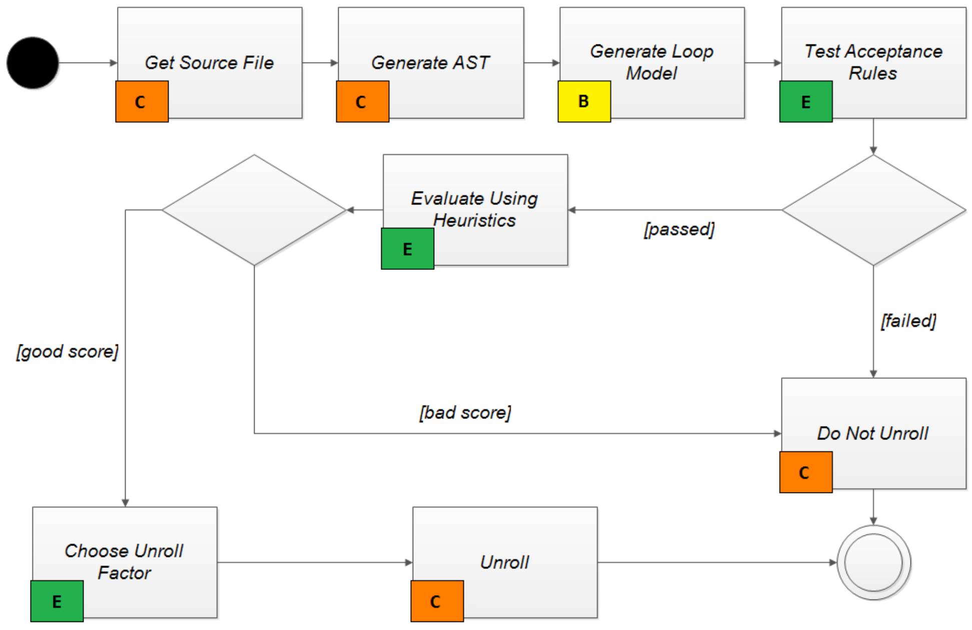


Figure 4.2: An activity diagram for the process used in this prototype. The different letters show how the different activities are performed by different software parts: *C* - Cetus; *E* - Evaluator Engine; *B* - Bridge.

This tool has available a wide range of source code analysis and source code transformations, none of which seem relevant in the context of this thesis.

4.3 Developed Software

There are two main pieces of software developed for this prototype. The Evaluation Engine receives a loop model and scores it according to the heuristics defined in a configuration file. The Bridge extracts information from the AST generated by Cetus and creates the loop model that is passed to the Evaluation Engine.

4.3.1 Evaluation Engine

The job of the Evaluation Engine is to evaluate a loop represented by an instance of the loop model. In order to do this, it needs to know what values the heuristics will take. It reads a configuration file to get this information. This file is a command line argument passed to the prototype. This engine has three main tasks. The first is to serve as a place where all the heuristics are stored. The second task is to, for a given loop, get the score of each heuristic and add it to the loop score. The third and final task is to calculate a suitable unroll factor.

```
public int evaluate(LoopModel loop)
{
  int score = 0;

  for( HeuristicGroup group : groups )
    score += groups[group].getGroupScore(loop);

  return score;
}
```

(a) The main evaluation method

```
public int getGroupScore(LoopModel loop)
{
  int groupScore = 0;

  for( Heuristic heuristic : heuristics )
    groupScore += heuristic.getScore(loop);

  return groupScore;
}
```

(b) The method that returns the score of a group

Figure 4.3: The two methods used to evaluate a loop: *evaluate* and *getGroupScore*.

After reading the configuration file, the engine will instantiate the heuristics with the correct values and become ready to evaluate a loop. The engine has two main methods that can be called separately. The first one, *evaluate*, evaluates a loop using the heuristics and the values that were loaded. The heuristics are stored in groups, allowing an easier organization and the possibility of using the intermediate scores for debug purposes. This method iterates over the groups of heuristics and adds the score of each group to the final loop score. The group score is given by another method, called *getGroupScore*. This method iterates over all the heuristics in the group and returns the sum of their individual scores for a given loop. Figure 4.3 show these two methods.

The second main method of the Evaluation Engine is used to suggest an unroll factor. It also receives a loop model and makes a suggestion based on the information contained on that model. Figure 4.4 shows this method which follows the flow described in the approach (Section 3.5, page 25).

There is no relation between the *evaluate* and *chooseUnrollFactor* methods. It could be possible that at the end of the evaluation method we could call the unroll method if the score was above a threshold. This is not the current approach. Instead, whatever function called the evaluation method (right now, the Bridge) should check the score and call the unroll method if the evaluation score is above its own threshold. This allows for some more flexibility and can separate the two methods if we only want to use one.

```java
public int chooseUnrollFactor(LoopModel loop)
{
  int fullFactor = loop.getIterationCount();

  if(loop.canRemoveArrayReferences())
    return fullFactor;

  int maxFactor = calculateMaxFactor(loop);

  if(loop.hasSmallIterationCount())
  {
    int difference = fullFactor - maxFactor;
    if(shouldFullyUnroll(difference))
      return fullFactor;
  }

  int smallestReuseDistance = loop.getSmallestReuseDistance();
  if(smallestReuseDistance > 0 && smallestReuseDistance > maxFactor)
  {
    int difference = smallestReuseDistance + 1 - maxFactor;
    if(shouldUseReuseDistance(difference))
      return smallestReuseDistance + 1;
  }

  return maxFactor;
}
```

Figure 4.4: The *chooseUnrollFactor* method.

4.3.2 Bridge

The Bridge is responsible for connecting the two sides of the prototype. We have the AST generated by Cetus on one side and the Evaluator Engine that needs a loop model on the other. The Bridge has three main tasks:

- Traverse the AST, gather all relevant information and build a loop model;
- Call the evaluation methods passing the created model;
- Create an annotation that informs the developer of the suggestion.

Initially we iterate over the AST and look for innermost *FOR* loops. When we find one, we extract all the information needed to build a model that represents that loop. We consult the loop header to get information about the iteration bounds, the step and the induction variable. Then, we get the statements inside the loop body. Finally, we need information about all the arrays that are accessed inside the loop body. This information is used to create a loop model.

When the model is completed, we pass it to the Evaluation Engine and ask for a evaluation of the loop. This is the second task of the Bridge, to call the evaluation methods. The Bridge holds the threshold value that indicates when a loop is considered a good candidate for Unrolling, which is 0 for this particular instance (i.e, a loop with a positive score is a good candidate). After calling the evaluation method for a loop and receiving its score, the Bridge compares the score with the threshold. If the score is greater, it calls the method responsible for choosing an unroll factor.

```
DepthFirstIterator<Traversable> iterator = new DepthFirstIterator<Traversable>(file);
try
{
  while(true)
  {
    ForLoop loop = (ForLoop) iterator.next(ForLoop.class);

    if(isInnerLoop(loop))
    {
      /* Get the needed information and create the LoopModel */
      LoopHeader head = getHeadInformation(loop);
      LoopBody body = getBodyInformation(loop, fileName, program);
      ArrayList<Array> arrays = getArraysInformation(loop, program);
      LoopModel loopModel = new LoopModel(head, body, arrays, fileName +"-"+ loopID);

      /* Create an Evaluator, pass it the model and evaluate the loop */
      EvaluatorEngine evaluator = new EvaluatorEngine(configFile);
      int score = evaluator.evaluate(loopModel);
      int factor = 0;
      if(score > 0)
        factor = evaluator.chooseUnrollFactor(loopModel);

      /* Annotate the result */
      createEvaluationAnnotation(score, factor, loop);

      loopID++;
    }
  }
}
catch (NoSuchElementException e)
{
  /* No more ForLoop elements in this file */
}
```

Figure 4.5: The code that performs the three tasks of the Bridge.

The final task of the Bridge is to inform the developer of the Evaluation Engine suggestion. We create an annotation on the IR tree with a comment that states whether the loop should be unrolled or not, and the unroll factor to use if we suggest to unroll. At the end of the program the IR tree is dumped to a new source file, which is the output of the prototype. This new source file has, for each inner loop, a comment of whether Loop Unrolling should be applied. Figure 4.5 shows the top level implementation of the Bridge, where the three tasks are performed.

The Bridge is implemented as an extension to Cetus, the compiler infra-structure used as a front-end that was described previously. This is done by overriding the method used to run passes and adding our own pass, which is run after the AST is generated.

4.3.3 Assumptions

In order to simplify the analysis some assumptions are made about the source code. These limit the usefulness of the prototype as they pose some constraints on the types of programs that can be analyzed. The following code characteristics are expected to be true:

- The step is calculated using a literal;
- The upper bound is declared using a literal;
- The initial condition of the loop is declared using a literal;
- When searching for array accesses, unary expressions are not considered.

While this means some programs will not be properly analyzed, it allows us to focus on more important issues and is acceptable considering this is a prototype.

4.4 Prototype Instance Values

This section shows a particular instance of the approach described before (see Chapter 3 on page 17). This includes the values for the heuristics and the metrics used to chose the unroll factor.

These values were especially created for the PowerPC architecture and specifically for the PowerPC 604 microprocessor. Because the heuristic values of this prototype were created for this processor, the prototype evaluation was also done using this processor (see Chapter 5 on page 43).

4.4.1 Heuristic Values

Table 4.1 shows the specific heuristic values for this instance of the presented approach. These heuristics were described in Section 3.4, page 21. This table shows, for each heuristic, the type of input, the possible values of the input and the respective scores.

These values were found empirically. First, through observation and experiment. During this stage we tried to look for effects of Loop Unrolling on several test cases and explain them using the characteristics present on their source code. We were able to see what heuristics had the best impact for this particular architecture. In a later stage we performed some exploration using random weights on each heuristic. With several heuristic configurations we were able to see how the changed heuristics compared to the base version. This stage allowed for some tuning of the values previously found.

4.4.2 Unroll Factor Values

This shows the value of *MaxInsts* for this instance and explains how it was found. This metric is the maximum number of assembly instructions inside a loop that does not cause the instruction cache performance to plummet. As of now, this is the only metric that is used to calculate the maximum unroll factor that does not hurt performance. This happens because out of all the Loop Unrolling negative effects only the instruction cache performance loss is being considered.

As was previously said (see Section 3.5, page 25) this metric is used to calculate *MaxFactor*:

$$MaxFactor = \left\lfloor \frac{MaxInsts}{LoopBodyInsts} \right\rfloor ,$$

where *LoopBodyInsts* is the number of assembly instructions on the loop body.

Table 4.1: The heuristic values for the PowerPC 604 microprocessor.

Heuristic	Input	Values	Scores
Small Iteration Count	Iteration Count	1 - 10	12
		11 - 20	6
		21 - 30	2
		31 - ∞	0
Data Reuse	Reuse Distance	1	20
		2 - 3	16
		4 - 10	8
		11 - 20	4
		21 - ∞	2
Same Scope Array	Possible?	Yes	28
		No	0
Loop Body Execution Time Relation	Execution Time Relation	1 - 3	4
		4 - 6	2
		7 - ∞	0
Number of Instructions	Instruction Count	1 - 17	0
		18 - 23	-4
		24 - 33	-8
		34 - 59	-16
		60 - ∞	-20

The value of this metric is **975**, meaning that when the body of a loop has 975 assembly instructions its instruction cache performance will collapse. Applying this value to the previous equation and taking as an example a loop with a body that has 40 instructions, the maximum unroll factor, *MaxFactor*, would be $\frac{975}{40} = 24.375$. After flooring the result we get an unroll factor of 24.

Once again, this value was found empirically, analyzing the results of the Loop Unrolling of several loops using different unroll factors. The goal of these tests was very simple, using a starting example we progressively increase the number of instructions on the loop body and see what is the unroll factor when the instruction cache performance starts to fall. The overall performance of the code is not meaningful as we are only interested in the effect on the cache. As an example consider test 3, whose results are in Figure 4.6. The instruction cache performance variation is represented by the green line. The point where we consider the instruction cache performance to plummet is marked with a circle and the unroll factor is **49**. The assembly code generated by the compiler had a loop body with 20 instructions, therefore, the number of instructions on the loop body when it is unrolled 49 times is approximately $49 \times 20 = 980$. This was done for 7 other tests and the results were averaged. Table 4.2 shows all the results. The column *Instructions* is the number of assembly instructions on the original loop body, the column *Unroll Factor* is the unroll

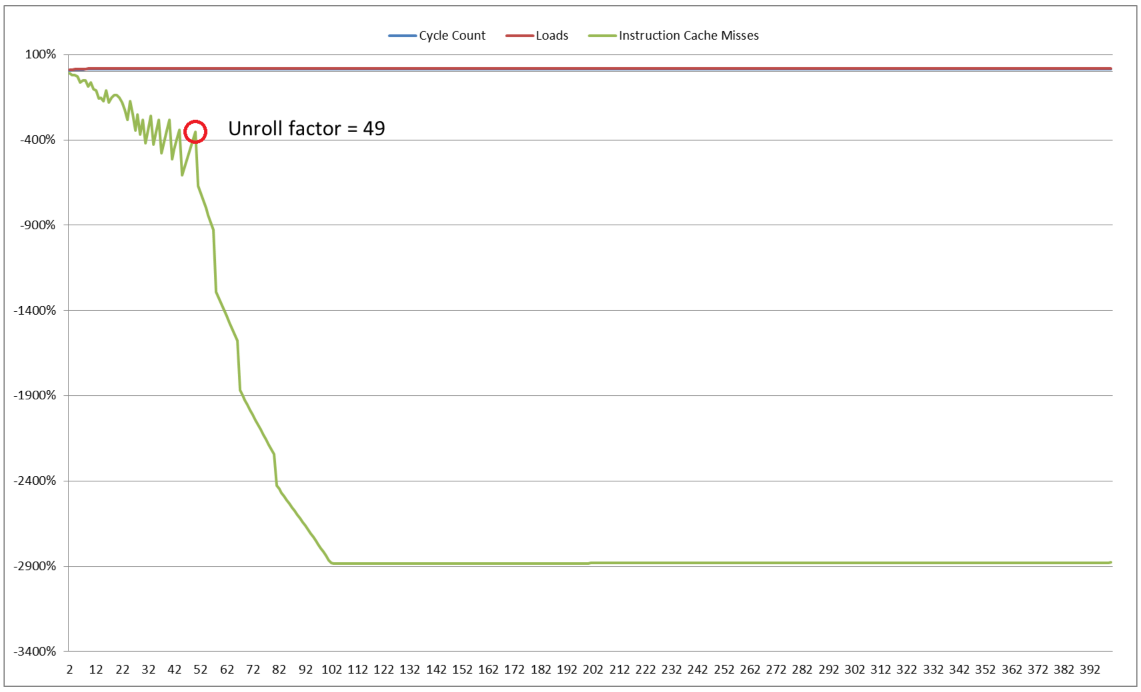


Figure 4.6: The results for Test 3. Unroll factors bigger than 49 (this factor is circled) cause the performance of the instruction cache to plummet.

factor from which the performance is considered to crash and the column *Max Instructions* is the number of instructions on the loop body for said unroll factor.

This value is completely dependent not only on the architecture but also on the machine where the code is going to be executed. This creates the need for an initial step of calibration in order to get the best value on the machine that this will be used on. Another disadvantage of this approach is that this requires assembly code analysis when we wanted to stay on a source code level. A future approach might consider blocks of source code instead of assembly instructions. Each of this blocks would represent a kind of operation and would account for the number of assembly instructions needed to execute.

Table 4.2: The results for the *MaxInsts* tests.

Test	Instructions	Unroll Factor	Max Instructions
1	9	130	912
2	13	78	937
3	20	49	980
4	12	98	982
5	14	77	1002
6	16	62	992
7	18	56	1008
8	19	52	988
Average			**975**

```
void doFIR(short* IN, short* OUT)
{
  int row, col, wrow, wcol;
  short K[] = {1, 2, 1,
               2, 4, 2,
               1, 2, 1};
  /* L1 */
  for ( row=0 ; row<348 ; row++ )
    /* L2 */
    for ( col=0 ; col<348 ; col++ )
    {
      int sumval = 0;

      /* L3 */
      for ( wrow=0 ; wrow<3 ; wrow++ )
        /* L4 */
        for ( wcol=0 ; wcol<3 ; wcol++ )
          sumval += IN[(row +wrow)*350+(col+wcol)] * K[wrow*3+wcol];

      sumval = sumval / 16;
      OUT[row * 350 + col] =  (short) sumval;
    }
}
```

Figure 4.7: Image Smooth Operation original code (ISO1) for images of 350x350 pixels.

Even though two other metrics (*MaxFull* and *MaxDistance*) were presented on the unroll factor suggestion process (see Section 3.5 on page 25), the tests done to find the values for this instance showed that these metrics were not used. Therefore, we didn't try to find appropriate values for them.

4.4.3 Evaluation Examples

We present two loop nests as examples and use this instance of the proposed heuristics to evaluate them. One of the examples had a good performance increase when transformed and the other had mixed results, where the performance change would be positive or negative depending on the unroll factor used.

The first example used is the one from the Image Smooth Operation test. The code for this loop can be seen in Figure 4.7. This example has a deep nest of loops. For this example only the two innermost loops (loops *L3* and *L4*) are considered. Initially we will evaluate the innermost loop (*L4*) and then we will evaluate the second inner loop (*L3*) considering that the inner loop is fully unrolled. When we are evaluating the second inner loop, the code looks like the code shown on Figure 4.8.

The second example that will be evaluated is the Gouraud test. The original code for this example is shown on Figure 4.9. When it was tested, several unroll configurations were used. This was a test that had somewhat good results with small unroll factors and bad results when the unroll factors were big. The best result obtained from the tests was an improvement of 7.93%,

```
void doFIR(short* IN, short* OUT)
{
  int row, col, wrow, wcol;
  short K[] = {1, 2, 1,
               2, 4, 2,
               1, 2, 1};
  /* L1 */
  for ( row=0 ; row<348 ; row++ )
    /* L2 */
    for ( col=0 ; col<348 ; col++ )
    {
      int sumval = 0;

      /* L3 */
      for ( wrow=0 ; wrow<3 ; wrow++ )
      {
        sumval += IN[(row +wrow)*350+col] * K[wrow*3];
        sumval += IN[(row +wrow)*350+(col+1)] * K[wrow*3+1];
        sumval += IN[(row +wrow)*350+(col+2)] * K[wrow*3+2];
      }

      sumval = sumval / 16;
      OUT[row * 350 + col] =  (short) sumval;
    }
}
```

Figure 4.8: Image Smooth Operation code after fully unrolling the innermost loop (ISO2).

which corresponds to an Unroll factor of 4. The worst result, obtained with an Unroll factor of 67, had about 3 times more cycles.

Table 4.3 shows how each individual heuristic evaluates and scores the example loops and Table 4.4 shows the final results of this evaluation.

The Image Smooth Operation example has two configurations. The first, which considers the innermost loop (*L4*), is called *ISO1* and the second, which considers the second inner (*L3*) loop after the innermost was fully unrolled, is called *ISO2*. According to the results from this evaluation both would be considered for Loop Unrolling, as they had positive scores. This is a good result seeing that both loops had a great performance improvement when fully unrolled. The *ISO2* configuration has a big score, which is an excellent result because fully unrolling this loop lead to a 79% performance gain. This score comes mainly from the fact that fully unrolling this loop will replace all the accesses to array *K* with its values, reducing the number of loads and eliminating the array. This is taken into consideration during the evaluation and, as shown before, gives a big score boost (see the *Same Scope Array* score). One thing to note is that the score on *ISO2* is higher than the score on *ISO1*, which makes sense considering that *ISO2* had a 79% improvement against the 33% improvement of *ISO1*.

From all the loops that were tested, the loop on Gouraud is one of the most difficult to evaluate because it has mixed results. For the most part it suffers a performance loss when unrolled. However, a performance gain of about 8% can not be overlooked especially when it results from such a simple transformation. This is a loop that could be considered a good candidate, but these

```
for ( i=0 ; i<SIZE ; i++ )
{
  r += rd;
  g += gd;
  b += bd;
  p[i] = (r & mask) + ((g & mask) >> 5) + ((b & mask) >> 10);
}
```

Figure 4.9: Gouraud original code.

heuristics evaluate it as unsuitable for Unrolling. Even though most unroll factors will cause a performance degradation, a carefully chosen factor can still improve the performance of this loop. Comparing this loop to the one evaluated on *ISO1* the results are positive. *ISO1* has a performance gain that is about four times bigger, so it has a better score.

4.5 Summary

This chapter presented the prototype that was implemented in order to evaluate our approach. This prototype uses an instance of the heuristic values that targets the PowerPC 604 processor. The prototype uses Cetus, a compiler infrastructure. After the AST is built, a piece of software, the Bridge, extracts all needed information and builds an instance of the loop model. This model is then passed to another piece of software, the Evaluation Engine. This engine, after reading a configuration of the heuristics from a file, is capable of evaluating the loop and suggesting an unroll factor when the loop score is above a threshold, which is zero in this prototype. The suggestion made by the Evaluation Engine is then communicated to the developer by means of a comment on the output source code. This comment appears before any loop that was analyzed and indicates whether the loop should be unrolled or not. In the cases where the suggestion is to perform Unrolling, it also indicates the chosen unroll factor. In order to properly evaluate a loop considering other target architectures, the heuristic values need to be instantiated.

Table 4.3: The scores given by each heuristic to the Image Smooth Operation and Gouraud examples.

	ISO1		ISO2		Gouraud	
Heuristic	Parameter	Score	Parameter	Score	Parameter	Score
Small Iteration Count	3	12	3	12	200	0
Data Reuse	no	0	no	0	no	0
Same Scope Array	no	0	yes	28	no	0
LB Execution Time Relation	6	2	10	0	8	0
Number of Instructions	31	-8	81	-20	33	-8
Final Score		6		20		-8

Table 4.4: Results for the Image Smooth Operation and the Gouraud examples.

Configuration	Score	Suggest for Unroll	Real Improvement
ISO1 (*L4*)	6	Yes	33%
ISO2 (*L3*)	20	Yes	79%
Gouraud	-8	No	7.93%

Prototype

Chapter 5

Experimental Results

This chapter uses a set of benchmarks to evaluate the approach presented in this dissertation. To compare the results obtained by our approach we collected data by exploring Loop Unrolling factors. We use the unroll factor that allows the best performance to compare with the unroll factor suggested by our prototype.

This chapter is organized as follows. Section 5.1 describes the setup used to test the benchmarks and how the data resulting from the tests was gathered. Section 5.2 presents the results obtained.

5.1 Setup

In order to evaluate our approach, a simple prototype was developed (see Chapter 4). This prototype uses heuristic values that target the PowerPC architecture and more specifically the PowerPC 604 [SDC94] processor. The prototype receives information about both the heuristics and the loops to analyze. The heuristics configuration file contains information about each of the heuristics used (i.e., their range and values). The loop information comes from the loop model that is built using information from the Abstract Syntax Tree (AST) created from the source file. The prototype then evaluates the loop, scores it accordingly, and suggests an unroll factor.

For each benchmark, we measured the performance gain obtained with every possible unroll factor. These data can be used to decide if a loop should be unrolled or not and, if so, what unroll factor causes the biggest performance improvement. To find these data, all possible unroll configurations for all benchmarks were run on the PSIM [Cag12] simulator. Each benchmark had a base source file with the loop to be tested and a configuration file that indicated how that loop would be unrolled. A script was then responsible for reading these files and creating all the source files needed, one for each unroll factor on the configuration file. These source files had a pre-processor directive on the loop that indicated how it would be unrolled. The next step was to unroll the loops. This was done using a tool that is part of Reflectc [Ref12, CDP$^+$11]. This tool

Table 5.1: The PSIM configuration used on the experimental evaluation.

Option	Value
Processor Model	PowerPC 604
Compliance Level	VEA
Endianness	Big-endian
Number of Processors	1
Emulated OS	NetBSD

uses specific pragmas to know how to unroll the loop. After this step there were several source files, one that has the original loop and the others that have the loop unrolled with different factors. The transformed source files were then compiled using GNU Compiler Collection (GCC) [Fre12], more specifically a GCC x86-PowerPC cross compiler. Finally, the resulting binaries were run on the PSIM simulator and the results were extracted using another script that compiled them on a single file. Whenever the results show a performance improvement on the form of a percentage, it represents the improvement (reduction) on the cycle count of the program.

The results obtained with the prototype were evaluated in three different ways. The first and most basic form of evaluation is to see if the suggestion made (i.e., to unroll or not) for each benchmark is the correct one. This allows us to know if the prototype is capable of finding whether Loop Unrolling will have a positive impact on the loop. The other two tests are only performed when there is a suggestion to unroll the loop. The second form of evaluation compares the suggested unroll factor with the unroll factor that leads to the best performance gain. By doing this we can see how good the prototype is at suggesting a good unroll factor. The third and final form of evaluation compares the performance obtained by unrolling the loop with the suggested unroll factor with the best performance gain possible. This is a good test because even though the prototype may suggest a different unroll factor, the difference between the two performance gains might be negligible, making the suggestion an efficient one.

All the tests were run under the LinuxMint 12 Linux distribution. This choice was made because a Linux based operating system was needed to use some of the tools. In order to apply Loop Unrolling automatically, a tool that is part of Reflectc was used. This tool is a source-to-source C compiler that was developed with the CoSy [Exp12] framework. The simulator that was used, PSIM, receives PowerPC binaries as input. To compile these binaries we needed a cross compiler. GCC was the choice since it can be built to target several different architectures.

PSIM was the used PowerPC simulator. It is bundled with GNU Debugger (GDB) but can be built separately (we followed this approach). It is capable of simulating three different PowerPC processors: 603, 603e and 604. It also supports all three levels of compliance as defined in [SSM94]. The exact configuration of the simulator is presented in Table 5.1.

PowerPC 604 is a Reduced Instruction Set Computer (RISC) microprocessor released in 1994, that was developed jointly by Apple, IBM and Motorola. *"The PowerPC 604 RISC microprocessor uses out-of-order and speculative execution techniques to extract instruction-level paral-*

Table 5.2: This table shows the suggestion made for each benchmark and whether that suggestion is the correct one. This suggestion can be to either apply Loop Unrolling or keep the loop in its original state. The incorrect suggestions are highlighted.

Benchmark	Score	Suggestion	Correct?
Dot Product	2	Apply	Yes
Gouraud	**-8**	**Keep**	**No**
Grid Iterate	-10	Keep	Yes
Vector Sum	2	Apply	Yes
ISO1	6	Apply	Yes
ISO2	20	Apply	Yes
FSD1	10	Apply	Yes
FSD2	**-6**	**Keep**	**No**

lelism" [SDC94]. It uses branch prediction and speculative execution as well as out-of-order execution and it has a six-stage superscalar pipeline. There are separate data and instruction caches, both with 16-Kbyte and four-way associativity.

The benchmarks chosen to evaluate this prototype represent the domain of application. In order to do this, real functions were used. These are functions from real life projects and applications. Achieving good results in these benchmarks is an indication of the usefulness of the prototype being tested. The source code of the benchmarks used can be seen in Appendix A (see page 57).

5.2 Results and Analysis

The results for the evaluation of each benchmark can be seen in Table 5.2. This table has the score of the benchmark, the suggestion made by the prototype, and whether that was a correct suggestion or not. The results are overall positive, with the prototype making 6 out of 8 correct suggestions. The wrong predictions, highlighted in the table, were for the Gouraud and FSD2 benchmarks. Gouraud is a difficult test case as it had mixed results. For small unroll factors it is possible to achieve modest performance gains but for unroll factors bigger than 4, there is a performance loss. In the worst case, the performance is about 2 times worse. The loop on this benchmark has a large number of instructions inside the loop body, which is heavily penalized by the evaluation heuristics. Furthermore this loop does not present any of the positive characteristics that the heuristics look for. There is no possibility of data reuse and there is no possibility of replacing any accesses to arrays. This means that the only benefit from Loop Unrolling on this case is the control structure overhead reduction.

Much like the Gouraud benchmark, the loop in FSD2 does not show characteristics that allow benefits other than overhead reduction. This means that it gets only a small number of points from the positive heuristics. The loop has a small body that multiplies two values and adds them to a sum variable. Although it is a simple body, it has a fairly high number of instructions and gets

Table 5.3: Comparing the suggested unroll factor and the optimal unroll factor (and their associated performance improvements) for each benchmark. The value *no* indicates no unroll to be done.

	Unroll Factor		Performance Improvement (%)		
Benchmark	Suggested	Optimal	Suggested	Optimal	Difference
Dot Product	57	58	37.25	37.27	0.02
Gouraud	no	4	0	7.93	7.93
Grid Iterate	no	no	0	0	0
Vector Sum	57	253	31.49	35.37	3.88
ISO1	3	3	33.30	33.30	0
ISO2	3	3	78.62	78.62	0
FSD1	8	8	6.33	6.33	0
FSD2	no	63	0	32.12	32.12

penalized because of that. Unlike Gouraud, the performance gain on this benchmark is positive for every possible unroll factor (within the test limits) which, by comparison, makes it a worse result.

Even though they are suggested for Unrolling, the benchmarks Dot Product, Vector Sum and ISO1 have low scores. This is a problem since they have very good performance improvements when unrolled. The loop bodies for these benchmarks are relatively small, especially on Dot Product and Vector Sum. Therefore, these are benchmarks that would benefit greatly from the overhead reduction that comes with Loop Unrolling. Like the two previously analyzed tests, Gouraud and FSD2, these have no obvious benefits other than the overhead reduction. What this means is that the positive heuristics reward them with little to no points. This seems to be affecting a large part of all the benchmarks that were tested (5 out of 8). One possible solution is to start the loop score with a positive value, instead of 0 (which is the current practice). By doing this it would be possible to account for the overhead reduction and results presented in Table 5.2 could be better.

For all the benchmarks that were suggested for Loop Unrolling we can compare the suggested unroll factor with the optimal unroll factor. In this context, the optimal unroll factor is the one that leads to the best performance improvement. Table 5.3 presents, for each benchmark, the unroll factor suggested by the prototype and the optimal unroll factor. This table also includes the benchmarks that were not considered good candidates for Loop Unrolling. These benchmarks have a value of *no*, which represents that the loops were not unrolled. For example, the Grid Iterate benchmark would be suggested as unsuitable for Unrolling and the tests confirm this, therefore, it has a value of *no* on both columns (*Suggested* and *Optimal*). One can also see, on the *Difference* column, the difference between the performance gains achieved with the optimal factor and the performance gains achieved with the suggested factor.

From the five loops that would be unrolled and therefore would have a suggested unroll factor, four of them were given a good unroll factor. On three of these loops, the optimal unroll factor was suggested, on another one the optimal factor was missed by one and on the final loop there is a big difference between the two factors.

ISO1, ISO2 and FSD1, the three loops whose suggested factor is the optimal one, all have a small iteration count. This means that it is probably better to fully unroll the loop. The evaluator prototype recognizes this and makes correct predictions.

The Dot Product benchmark has a really good result. Even if the suggested factor is not the optimal one, the difference is only **1**. The Vector Sum benchmark, however, which is similar to Dot Product, does not manage to get a similar result. The difference between the optimal factor and the suggested one is **196**. This happens because the only metric used to suggest a good unroll factor is the number of instructions and even though the loops have different bodies, they have a similar number of instructions which leads to the same suggested unroll factor.

There were two benchmarks whose suggested unroll factor is not the optimal one, Dot Product and Vector Sum. Even if the suggested unroll factor is not close to the optimal one, this difference can be negligible if the difference between the performance gain obtained with both is minimal (see Table 5.3). This is the case with these two benchmarks. For Dot Product, even though there is a difference of one between the optimal factor and the suggested factor, the difference in performance gain is just 0.02%. This difference is not significant and as such the suggested unroll factor can be classified as a good one. The results on Vector Sum, while not as good, are still positive. Despite having a completely different unroll factor, the difference of the performance gain is only of 3.88% (on an optimal performance gain of 35.37%). Once again, while the suggestion fails to find the optimal unroll factor, the overall result is good.

5.3 Summary

To evaluate the prototype, a series of tests were conducted on a simulated PowerPC 604, a RISC microprocessor with branch prediction, speculative execution and out-of-order execution. The tests were created to find, for a set of benchmarks, what was the unroll factor that would lead to the best performance improvement. With this information it was possible to know if the evaluator is capable of accurately predicting the impact of Loop Unrolling on a certain loop and how good the suggested unroll factor is.

Overall the results are good. The evaluator was capable of making the right prediction on the majority of the tested benchmarks. For those benchmarks that would be suggested for Loop Unrolling, the suggested unroll factor is also good. Only one of the benchmarks was suggested an unroll factor that was far from the optimal one and even in this case the difference in performance improvement is not too significant.

Loops that do not exhibit obvious advantages (e.g., data reuse) are not well evaluated. This happens because the overhead reduction caused by Loop Unrolling is not correctly taken into account by the heuristics. Because of this, certain benchmarks that would benefit from this transformation were not suggested for Unrolling while others had low scores. If one was to compare the scores of the loops and try to establish a correlation between the score and the performance improvement, these scores fail to show how great the impact of the transformation is.

The only metric that is used to suggest an unroll factor is the number of assembly instructions present on the loop body. While the results of the suggested unroll factors are acceptable, there is at least one example where two loops whose optimal unroll factors are completely different were suggested the same unroll factor just because they had a similar number of instructions. While this metric seems capable of finding suitable unroll factors, better results could be achieved if other factors would be taken in consideration.

Chapter 6

Conclusions

Performance is an important factor on every application, being its importance more noticeable in embedded computing applications. One possible way to improve performance is to use source code transformations. These can have a big impact if applied to loops, where programs spend most of their execution time. There are many loop transformations, all of them with different approaches and goals. One of the most used loop transformations is Loop Unrolling, which is the focus of this dissertation.

Because predicting the impact of any transformation can prove rather challenging, there is not an universal approach. As a consequence, very frequently, the developer has to apply the transformation manually. As this can be an error-prone process, we need a tool that helps the developer by suggesting loops that represent good optimization opportunities. Such a tool would still keep the developer in control but would give him the safety of knowing that applying a transformation would be beneficial. This dissertation addresses this problem, combining a heuristic-guided approach with Loop Unrolling in order to suggest good loops to optimize.

6.1 Main Conclusions

This dissertation presented an approach for the problem of performance optimization using heuristics, source code transformations and suggestions. It is an approach that can be used to guide a developer in the process of optimizing but that makes sure that he still has the final decision on what happens to his code. This is useful when the knowledge of the user is needed to make sure the transformation will be beneficial. The proposed method suggests good Loop Unrolling candidates but it acknowledges that it might not have the perfect information every time and that the developer insight should be taken into account. By making only a suggestion the developer can provide his input and ultimately decide if the transformation should be applied.

Our approach relies on a set of heuristics to evaluate loops as candidates for Loop Unrolling. While the heuristic values are closely related to the machine where the program will run, the heuristics themselves depend mainly on the source code and, we believe, can possibly be used on different architectures. Two of the five heuristics presented, *Number of Instructions* and *Loop Body Execution Time Relation*, use information that is not readily available on the source code. These two heuristics use assembly information, which obviously depends on code characteristics.

To validate and evaluate the approach presented in this dissertation we created a prototype that used an instance of the heuristics for a specific processor, the PowerPC 604. This prototype was tested using a group of 8 benchmarks extracted from real life applications. The prototype was able to correctly suggest Loop Unrolling for 6 of the 8 benchmarks used. Out of these 6 benchmarks, 5 were considered good candidates for Loop Unrolling and were given suitable unroll factors. The suggested factor was close or equal to the optimal factor on 4 of these. On the other one, even though there is a big difference in the unroll factor, the difference on the performance gains achieved is not significant (31.49% with the suggested factor and 35.37% with the optimal one).

Even though the results are positive, this approach has some shortcomings. Currently, only innermost *FOR* loops where the iteration count is known at compile time are considered. Even though these loops are used by a large number of embedded applications, this approach may lose some improvement opportunities by not considering other types of loops.

The unroll factor is a major part of Loop Unrolling. If an unsuitable factor is used, the transformation can easily cause a performance loss. The method presented to choose an unroll factor only considers the increasing number of instructions and its effect on the cache. While it manages to suggest suitable unroll factor, other information could be used in order to improve this suggestion.

Finally, we can also conclude, after analysing the experimental results, that the overhead reduction could be better evaluated. It is one of the main advantages of Loop Unrolling and, therefore, is of great importance. Some of the benchmarks where no other positive characteristics were found have lower scores than expected. This means that the heuristic responsible for modelling the overhead reduction, if improved, could lead to better evaluation results.

6.2 Future Work

As referred before, the approach presented in this dissertation can be improved in a number of ways. Two of the five heuristics presented rely on information that can not be directly extracted from source code. These are the heuristics that account for the increasing number of instructions and the relation between the execution time of the body and the control structure. These heuristics count the number of assembly instructions that the compiler generates for the loop body and use this information to evaluate the loop. It would be interesting to use a source level alternative that would make the analysis only source code dependent.

A possible solution is to consider code blocks. These represent blocks of instructions that execute simple tasks. Consider a loop that loads two consecutive values from an array and that stores their product on another array. As seen before, Loop Unrolling exposes several iterations and allows for data reuse. Exposing at least two iterations of that loop would cause one value from one iteration to be used on another, which results in one less load. If we were to consider code blocks and one of them was a memory load, after unrolling the loop twice, the second iteration would lose a load block as it would be able to reuse data previously loaded. This would not only make the analysis source code dependent, but would account for an effect that is not currently considered, which is the fact that some instructions will be removed after Loop Unrolling is applied.

Another change that would have a big impact is a better way of finding a good unroll factor. Right now, only one metric is used: the number of instructions. As was shown before, this is not enough and needs to be expanded. Even though this is an area with room for improvements, it can be a hard challenge. One possible advance is the usage of the code blocks mentioned earlier. Using that model we could make a better prediction of when the instruction cache would overflow and use it to suggest a more accurate unroll factor.

It would be interesting to use a machine learning approach to find new, better heuristics and, eventually, improve the existing ones. Even though most heuristics will be somewhat portable[1] as they look at source code, their instance values for a specific machine are not. There is a need for fine tuned, specialized values. Imagining the prototype developed as a full application, we can think of a calibration step that, before running the application for the first time, would find the best possible values for the heuristics on that particular machine. Every source code analysis made from then on, would inherently take machine and architecture information into account.

The developed prototype could also be improved. The prototype is a standalone tool, that can only be used to suggest loops that are good candidates for Loop Unrolling. The possibility of integrating it with other tools seems rather appealing. Imagine the current prototype connected to a source-to-source transformation tool. The prototype suggests a certain loop for full unroll. The developer is presented this suggestion and is asked if he wants to transform the loop. After analyzing the source code he confirms the benefits of applying this transformation and chooses to unroll the loop. The loop (or properly annotated source file) would then be passed to the transformation tool which would apply Loop Unrolling. The developer still decides on whether the transformation is applied or not, but the transformation process is automated, relieving the developer of a tedious and error-prone activity. This could also be possible using the current software. The chosen compiler infrastructure, Cetus, can be used to transform source code. Cetus makes it possible to transform the high level intermediate representation that will then be written as C source.

If the previous integration with (or development of) a transformation tool was completed, we could consider using this prototype in an iterative fashion, i.e., using the output of an evaluation as the input of another. The current approach only considers innermost loops, which means that sometimes we do not consider good transformation opportunities. If an inner loop is fully unrolled, which is often the case when there is a small iteration count, the loop in the level immediately above could also be tested. This could lead to an increase in the transformation opportunities while still retaining the simple, innermost loop only approach.

[1]In this context *portable* means that they could be applied to different architectures.

References

[ABC$^+$06] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[Cag12] Andrew Cagney. PSIM - Model of the PowerPC Architecture, March 2012. Please see: `http://sources.redhat.com/psim/`.

[CD04] João M. P. Cardoso and Pedro C. Diniz. Modeling Loop Unrolling: Approaches and Open Issues. In Andy D. Pimentel and Stamatis Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation*, volume 3133 of *Lecture Notes in Computer Science*, pages 224–233. Springer Berlin - Heidelberg, Berlin, Heidelberg, 2004.

[CDP$^+$11] João M. P. Cardoso, Pedro C. Diniz, Zlatko Petrov, Koen Bertels, Michael Hübner, Hans Someren, Fernando Gonçalves, José Gabriel F. Coutinho, George A. Constantinides, Bryan Olivier, Wayne Luk, Juergen Becker, Georgi Kuzmanov, Florian Thoma, Lars Braun, Matthias Kühnle, Razvan Nane, Vlad Mihai Sima, Kamil Krátký, José Carlos Alves, and João Canas Ferreira. REFLECT: Rendering FPGAs to Multi-core Embedded Computing. In João M. P. Cardoso and Michael Hübner, editors, *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*, pages 261–289. Springer New York, 2011.

[DBM$^+$09] C. Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, R. Eigenmann, and S. Midkiff. Cetus: A Source-to-Source Compiler Infrastructure for Multicores. *Computer*, 42(12):36–42, December 2009.

[DC11] Pedro Diniz and João Cardoso. Code Transformations for Embedded Reconfigurable Computing Architectures. In João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 322–344. Springer Berlin - Heidelberg, 2011.

[DH79] J. J. Dongarra and A. R. Hinds. Unrolling Loops in FORTRAN. *Software: Practice and Experience*, 9(3):219–226, 1979.

[DJ01] Jack W. Davidson and Sanjay Jinturkar. An Aggressive Approach to Loop Unrolling. Technical report, University of Virginia, Charlottesville, VA, USA, 2001.

[DMBW08] Ozana Dragomir, Elena Moscu-Panainte, Koen Bertels, and Stephan Wong. Optimal Unroll Factor for Reconfigurable Architectures. In Roger Woods, Katherine

Compton, Christos Bouganis, and Pedro Diniz, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4943 of *Lecture Notes in Computer Science*, pages 4–14. Springer Berlin - Heidelberg, 2008.

[Dra11] O.S. Dragomir. *K-loops: Loop Transformations for Reconfigurable Architectures*. PhD thesis, TU Delft, Faculty of Elektrotechniek, Wiskunde en Informatica, Delft, Netherlands, 2011.

[Exp12] Associated Compiler Experts. CoSy compiler development system, May 2012. Please see: `http://www.ace.nl/compiler/cosy.html`.

[Fre12] Free Software Foundation. GCC, the GNU Compiler Collection, March 2012. Please see: `http://gcc.gnu.org/`.

[FS95] Yoav Freund and Robert Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In Paul Vitányi, editor, *Computational Learning Theory*, volume 904 of *Lecture Notes in Computer Science*, pages 23–37. Springer Berlin - Heidelberg, 1995.

[FS99] Y. Freund and R. Schapire. A Short Introduction to Boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771–780, 1999.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 3rd edition, 2003.

[KKF97] A. Koseki, H. Komastu, and Y. Fukazawa. A Method for Estimating Optimal Unrolling Times for Nested Loops. In *Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks*, pages 376–382, Washington, DC, USA, December 1997. IEEE Computer Society.

[Kob84] M. Kobayashi. Dynamic Characteristics of Loops. *IEEE Transactions on Computers*, C-33(2):125–132, 1984.

[MBQ02] Antoine Monsifrot, François Bodin, and René Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In Donia Scott, editor, *Artificial Intelligence: Methodology, Systems, and Applications*, volume 2443 of *Lecture Notes in Computer Science*, pages 41–50. Springer Berlin - Heidelberg, Berlin, Heidelberg, 2002.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[mip] *MIPSPro 7 Fortran 90 Commands and Directives Reference Manual, 007-3696-03*.

[MKS94] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A System for Induction of Oblique Decision Trees. *Journal of Artificial Intelligence Research*, 2(1):1–32, August 1994.

[PCC09] Martin Palkovic, Francky Catthoor, and Henk Corporaal. Trade-offs in Loop Transformations. *ACM Transactions on Design Automation of Electronic Systems*, 14(2):22:1–22:30, April 2009.

[Ref12] Reflect. Reflect Project, February 2012. Please see: `http://www.reflect-project.eu/`.

[SA05] M. Stephenson and S. Amarasinghe. Predicting Unroll Factors Using Supervised Classification. In *International Symposium on Code Generation and Optimization, 2005. CGO 2005.*, pages 123–134, March 2005.

[Sar00] Vivek Sarkar. Optimized Unrolling of Nested Loops. In *Proceedings of the 14th international conference on Supercomputing*, ICS ’00, pages 153–166, New York, NY, USA, 2000. ACM.

[SDC94] S.P. Song, M. Denman, and J. Chang. The PowerPC 604 RISC Microprocessor. *IEEE Micro*, 14(5):8–17, October 1994.

[SSM94] Ed Sikha, Rick Simpson, and Cathy May. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann, June 1994.

[TVVA03] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. Compiler Optimization-Space Exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 204–215, March 2003.

[Uni11] Purdue University. The Cetus Project, December 2011. Please see: `http://cetus.ecn.purdue.edu/`.

[VLCV01] J. Villarreal, R. Lysecky, S. Cotterell, and F. Vahid. A Study on the Loop Behavior of Embedded Programs. Technical report, Department of Computer Science and Engineering, University of California, Riverside, CA, USA, 2001.

[WMC96] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining Loop Transformations Considering Caches and Scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 274–286, Washington, DC, USA, 1996. IEEE Computer Society.

[Wol95] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman, Boston, MA, USA, 1995.

[WS87] Shlomo Weiss and James E. Smith. A Study of Scalar Compilation Techniques for Pipelined Supercomputers. In *Proceedings of the second international conference on Architectual support for programming languages and operating systems*, ASPLOS-II, pages 105–109, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[WS90] D. Whitfield and M. L. Soffa. An Approach to Ordering Optimizing Transformations. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, PPOPP ’90, pages 137–146, New York, NY, USA, 1990. ACM.

[ZCS03] Min Zhao, Bruce Childers, and Mary Lou Soffa. Predicting the Impact of Optimizations for Embedded Systems. In *Proceedings of the 2003 ACM SIGPLAN conference on Languages, compilers, and tools for embedded systems*, LCTES ’03, pages 1–11, New York, NY, USA, 2003. ACM.

REFERENCES

Appendix A

Benchmarks

This appendix shows the source code for all the benchmarks used to evaluate the prototype that was developed. In these figures only the loop is presented. When there is a loop nest, only the innermost loop is considered.

```
for ( i=0 ; i<SIZE ; i++ )
{
  sum += x[i] * y[i];
}
```

Figure A.1: Dot Product benchmark.

```
for ( i=0 ; i<SIZE ; i++ )
{
  r += rd;
  g += gd;
  b += bd;
  p[i] = (r & mask) + ((g & mask) >> 5) + ((b & mask) >> 10);
}
```

Figure A.2: Gouraud benchmark.

```
for ( it=0 ; it<ITER_STEPS_NUM ; it++ )
{
  for ( i=1 ; i<X_DIM-1) ; i++ )
  {

    for ( j=1 ; j<(Y_DIM-1) ; j++ )
    {
      for ( k=1 ; k<(Z_DIM-1) ; k++ )
      {
        val = obstacles[i][j][k];

        if (val == 1)
        {
          potential[i][j][k] = POTENTIAL_ZERO;
        }
        else
        {
          if (val == -1)
          {
            potential[i][j][k] = POTENTIAL_ONE;
          }
          else
          {
            acc  =  (accType)potential[i-1][j][k] +
                (accType)potential[i+1][j][k] +
                (accType)potential[i][j-1][k] +
                (accType)potential[i][j+1][k] +
                (accType)potential[i][j][k-1] +
                (accType)potential[i][j][k+1];

            potential[i][j][k] = CACM_FIX_CORRECTION(acc * POTENTIAL_SIXTH);
          }
        }
      }
    }
  }
}
```

Figure A.3: Grid Iterate benchmark.

```
for ( i=0 ; i<SIZE ; i++ )
{
  prod = x[i] * x[i];
  sum += prod;
}
```

Figure A.4: Vector Sum benchmark.

```
for( row=0 ; row<348 ; row++ )
{
  for( col=0 ; col<348 ; col++ )
  {
    int sumval = 0;

    for( wrow=0 ; wrow<3 ; wrow++ )
    {
      for( wcol=0 ; wcol<3 ; wcol++ )
      {
        sumval += IN[(row +wrow)*350+(col+wcol)]*K[wrow*3+wcol];
      }
    }

    sumval = sumval / 16;
    OUT[row * 350 + col] =  (short) sumval;
  }
}
```

Figure A.5: ISO1 benchmark.

```
for( row=0 ; row<348 ; row++ )
{
  for( col=0 ; col<348 ; col++ )
  {
    int sumval = 0;

    for( wrow=0 ; wrow<3 ; wrow++ )
    {
      sumval += IN[(row +wrow)*350+col]*K[wrow*3];
      sumval += IN[(row +wrow)*350+(col+1)]*K[wrow*3+1];
      sumval += IN[(row +wrow)*350+(col+2)]*K[wrow*3+2];
    }

    sumval = sumval / 16;
    OUT[row * 350 + col] =  (short) sumval;
  }
}
```

Figure A.6: ISO2 benchmark.

```
for( i=0 ; i<64 ; i++ )
{
  y[i] = 0.0;

  for( j=0 ; j<8 ; j++ )
  {
    y[i] += z[i+64*j];
  }
}
```

Figure A.7: FSD1 benchmark.

```
for( i=0 ; i<32 ; i++)
{
  s[i]=0.0;

  for( j=0 ; j<64 ; j++ )
  {
    s[i] += m[32*i+j] * y[j];
  }
}
```

Figure A.8: FSD2 benchmark.