SUGGESTING LOOP UNROLLING USING A HEURISTIC-GUIDED APPROACH

Pedro Pinto

Dissertation done under the supervison of *Prof. João Cardoso*
at *FEUP*

1. Motivation

The development of embedded applications typically faces memory and/or execution time constraints and as such, performance should be considered during development. A way of improving performance is to use code transformations, which is already done today by advanced compilers and optimization tools. However, there are cases where these code transformations can not be automatically applied and the developer must implemented them manually.

There is a need for a tool that can help developers make confident decisions about when and how to make these transformations. The developer needs to feel safe when applying a code transformation, knowing that it will lead to a performance improvement. Because, ultimately, it is the user that is going to transform the code, this tool should be able to help while keeping the developer in control. A possible solution is to make suggestions about what transformations would be beneficial, giving a clear indication of where to go.

2. Goals

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. This approach will use a set of heuristics and source code information to make a prediction of whether Loop Unrolling will achieve a performance improvement.

Because this approach needs to be used by a developer it only makes sense to work on a high level and use a source code transformation. Loop Unrolling is a good fit as it is widely know and has been used for decades. It is easily applicable and always legal to use.

This approach should be translated to a prototype so that it could be validated and evaluated. The heuristics found should be instantiated to target a specific architecture that is meaningful in the context of embedded systems and applications.

3. Description

We first present the proposed approach and then the heuristics used in this approach. Later, we show the prototype that was developed as an instance to a specific processor and finally present the results of the prototype evaluation and analyze those results.

3.1. Proposed Approach

The approach consists of a four step process. Initially, the loop being analyzed is converted to a model that holds its relevant information. This model is used throughout the other three steps. The loop model needs information about the loop iteration bounds, step and induction variable. It also needs to know how many instructions the loop body has and if there are arrays being accessed.

The second step uses the model to test the loop against a set of acceptance rules. The main goal of these rules is to guarantee that the loop iteration count is known at compile-time. If this is not possible, this test will fail and the loop is discarded as a bad candidate.

If the loop is in compliance with the acceptance rules it can then be evaluated, which is the third step. The loop has a score, that is initially 0. This score will change as the loop is evaluated by the proposed heuristics. If, at the end of the evaluation, the score is above a threshold the loop is considered a good candidate and will advance to the fourth and final step.

At this point the only thing left to do is chose a suitable unroll factor. Because the loop was already tested twice, it is safe to assume that it will benefit from Loop Unrolling. This allows for a very simple strategy to chose the unroll factor. We can pick the biggest unroll factor that will not cause instruction cache thrashing.

3.2. Heuristics

There are five different heuristics in our proposed approach. The first four look for characteristics that, when found, will likely lead to a performance improvement.

Small Iteration Count This heuristic rewards loops that execute smaller numbers of iterations. When a loop has a small number of iterations it is less likely to cause instruction cache thrashing. There is also little danger when applying full Loop Unrolling.

Data Reuse This heuristic looks for opportunities to reuse data. If the loop shares data across iterations, Loop Unrolling might, by exposing those iterations together, allow the compiler to reuse that data. The distance between iterations that use the same data is called reuse distance. This heuristic gives a higher score the shorter this distance as smaller reuse distances allow to reuse more values with the same unroll factor.

Same Scope Array If the loop iterates over an array of constants and it is possible to see its declaration, then Loop Unrolling might enable the replacement of the array accesses with their values. If there is such an array, after full Unrolling the loop and applying transformations like Constant Propagation and Constant Folding, the compiler might remove the array accesses and replace them with the constant array values.

Loop Body Execution Time Relation One of the main advantages of Loop Unrolling is that, independently of everything else, it can reduce the control structure overhead. Every iteration executes a control structure, responsible for updating the induction variable and correctly terminating the loop. This transformation reduces the number of iterations, which in turn reduces the time spent executing this control code.

Number of Instructions Loop Unrolling increases the number of instructions inside the loop body, which can have a negative effect on the instruction cache performance. The bigger the unroll factor, the more instructions inside the loop body. This situation can lead to an increase in the number of cache misses, which will result in a performance loss that can overshadow whatever benefits Loop Unrolling brought.

3.3. Prototype

A prototype was developed to target the PowerPC architecture and more specifically the PowerPC 604 processor. The heuristics and metrics used to suggest Loop Unrolling and to choose a suitable unroll factor were instantiated for this processor. Through empirical observation and experiments it was possible to find values that suit the target processor.

The prototype makes use of an existing source-to-source compiler infra-structure, Cetus. Cetus is mainly used as front-end, whose main task is to translate source code to an Abstract Syntax Tree (AST). This AST is used to create, for each candidate loop, a model that is fed to an evaluation engine. This engine has full knowledge of the heuristics and their values and can evaluate the loop.

Cetus is used once again to make the suggestion resulting from the evaluation available to the developer. By transforming the AST it is possible to create a comment before each analyzed loop with the suggestion. Cetus will output the contents of the AST back to a source code file, where the user can see the result of the evaluation.

3.4. Results

The results are, overall, positive. From the 8 evaluated benchmarks, 6 were correctly suggested for Loop Unrolling. Tab. 1 shows the evaluation of each benchmark. It is possible to see the evaluation score, the suggestion (to unroll or to keep) and whether this suggestion is correct.

Tab. 1 – The suggestion made for each benchmark.

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

Tab. 2 presents the unroll factor suggestion. It is possible to see, for each benchmark that was suggested for Unrolling, the suggested unroll factor and the associated performance improvement. For comparison it is also possible to see the optimal unroll factor, i.e., the unroll factor that leads to the largest performance improvement. On 4 of the 5 benchmarks, the unroll factor is either equal or close to the optimal factor and on the other benchmark, there is a performance difference of just 3.88%.

Tab. 2 – Comparing the suggested unroll factor and the optimal unroll factor (and their associated performance improvements) for the benchmarks suggested for Unrolling.

	Unroll Factor		Improvement	
Benchmark	Suggested	Optimal	Suggested	Optimal
Dot Product	57	58	37.25%	37.27%
Vector Sum	57	253	31.49%	35.37%
ISO1	3	3	33.30%	33.30%
ISO2	3	3	78.62%	78.62%
FSD1	8	8	6.33%	6.33%

4. Conclusions

This dissertation presents a different approach to the problem of performance optimization. It does so by using heuristics, a source code transformation and suggestions. The results show that it can help the developer by indicating which loops to unroll and providing a suitable unroll factor. Even tough their values will always be adjusted and tuned for a specific architecture, the heuristics presented can be used for any architecture as they rely mostly on source-level characteristics.

This approach has some shortcomings. It considers only innermost *FOR* loops whose iteration count is known at compile-time. Therefore, only addresses a small part of all the possible loops that could be transformed. Furthermore, the strategy used to chose an unroll factor could use a big improvement. In its current state it only accounts for the number of instructions on the loop body and their effect on the instruction cache. The overhead reduction, one of the main advantages of Loop Unrolling, is not properly accounted for, even though there is an heuristic that targets it (Loop Body Execution Time Relation).