



IST ACOTES Project
Deliverable D4.3
Cost model
implementation in GCC4
interacting with ASM

IST-034869

Public

Project Number	:	IST-034869
Project Title	:	Advanced Compiler Technologies for Embedded Streaming
Deliverable Type	:	Prototype

Deliverable Number	:	D4.3
Title of Deliverable	:	Cost model implementation in GCC4 interacting with ASM
Nature of Deliverable	:	Public
Internal Document Number/version	:	acotes-d4.3-final
Contractual Delivery Date	:	31 May 2008
Actual Delivery Date	:	8 September 08
WP(s)	:	WP4 – Fine grain data-parallelism extraction and low level optimisation
Author(s)/Affiliation	:	Ayal Zaks, Dorit Nuzman, Razya Ladelsky

Abstract

This deliverable describes the prototype of the cost model, which was implemented in GCC4 and interacts with the ASM, governing the application of the automatic vectorizer to maximize the exploitation of fine-grain data-level parallelism at the task level.

Keyword list

Cost model, vectorization, GCC, ASM

Table of Contents

Table of Contents.....	2
1 Introduction.....	3
2 The Cost Model	4
2.1 Estimating Minimum Number of Iterations.....	4
2.2 Profitability equation	5
2.3 Interaction with Abstract Streaming Machine (ASM)	5
References	6

1 Introduction

The goal of the ACOTES project is to substantially improve the programmer's productivity in developing applications that process massive streams of data on programmable, parallel embedded architectures. The target application areas of ACOTES are video processing and advanced radio communications for consumer systems. Work Package (WP) 2 of ACOTES defined a Streaming Programming Model (SPM) and an Abstract Streaming Machine (ASM). The ASM itself consists of two parts: the first part deals with the inter-core/tile/cluster characteristics of the target platform, including coarse-grained Data Level Parallelism (DLP) features, and serves as an input to the high-level analyses of WP2 and optimizations of WP3; the second part deals with the capabilities of the target platform at the level of each individual core/tile/cluster (including fine-grained DLP), and serves as an input to the low-level optimizations and primarily auto-vectorization (WP4).

This deliverable presents the first prototype of a cost-model framework, incorporated into the automatic vectorization engine of GCC4, which aims to best exploit fine-grain DLP of individual cores according to the ASM of the underlying platform. Interactions between the cost-model and high-level optimizations of WP3 have been discussed in Deliverable D4.2, and have been further advanced by providing a set of API's in GCC4 [5]. The present deliverable focuses on the implementation of the cost model itself, as it was implemented in GCC4, and its interaction with the ASM.

The interested reader is encouraged to learn more about the prototype by examining the documented code of GCC's main development branch, freely available at <http://gcc.gnu.org/svn.html>; indeed this document does not come to replace the code and its detailed documentation nor duplicate it, but rather to supply an overview of the implementation thereby making the code hopefully more accessible to those interested.

The structure of the document is as follows: Section 2 describes the overall cost model with its main entry points and source files; subsection 2.1 focuses on estimating the minimum number of iterations for vectorization to be profitable; subsection 2.2 presents the general profitability equation, and subsection 2.3 discusses the interaction between the cost model and the ASM. A list of pointers to key code drops and discussion threads are provided at the end.

2 The Cost Model

When analyzing a loop, after GCC4 determines that it can be vectorized, we compare the expected profitability (in terms of total number of cycles) of the original scalar version with that of the vectorized version we're about to generate. This comparison is the last check that may prevent a loop from being vectorized, and it takes place in `tree-vect-analyze.c/vect_analyze_loop()` during the last call to `vect_analyze_operations()`, which invokes `vect_estimate_min_profitable_iters()` (found in `tree-vect-transform.c`) on the currently examined loop. Several costs associated with the different vectorization steps are computed and recorded throughout the `tree-vect-analyze.c/vect_analyze_loop()` phase, and are then used by `vect_estimate_min_profitable_iters()`.

2.1 Estimating Minimum Number of Iterations

The `vect_estimate_min_profitable_iters()` function is the crux of the cost model, performing an in-depth architecture-dependent analysis of the loop structure and contents. The underlying assumption is that the kernel of the loop usually executes faster if vectorized than if not, but that associated overheads may hinder the vectorized version, diminishing its speedup compared to the original scalar version, and more so for loops that iterate a small number of times. Indeed, if the number of iterations (N) of a loop is smaller than its vectorization factor (VF), there is no potential for speeding it up using vectorization; on the contrary, vectorizing such a loop may only slow down its execution due to additional preparatory actions and checks. Furthermore, even if N is larger than VF , the number of iterations of the vectorized loop, although positive, may not suffice to out-weigh the overheads incurred by vectorization. The `vect_estimate_min_profitable_iters()` function quantifies the overheads associated with vectorization, as well as the potential expected speedup of executing the loop in vectorized form compared to scalar form.

The `vect_estimate_min_profitable_iters()` function returns one number, indicating the minimum number of iterations required for the vector version of the loop to be faster relative to the expected execution speed of the scalar version of the loop. This number can be negative, to indicate that the loop should never be vectorized; this holds, for example, if the kernel of the loop executes slower if vectorized than if not. This number can be zero, to indicate that the loop should always be vectorized; this holds, for example, if there are no vectorization overheads. (This option is used, by the way, when disabling the cost model.) Otherwise, this number is at-least VF (or rather $VF-1$ due to the comparison used later). The user can supply a more conservative lower bound, for example to avoid vectorizing loops whose iteration count is less than twice VF , even if `vect_estimate_min_profitable_iters()` permits it.

Once the value x of `vect_estimate_min_profitable_iters()` is computed, it is used in one of two ways: if the number of loop iterations N is known at compile time, the loop is vectorized if and only if $N > x$. If the number of loop iterations N is not known at compile time, the vectorizer will consider inserting a runtime check comparing N with x , branching to a scalar version of the loop if $N < x$ and otherwise falling through to a vectorized version of the loop. To minimize performance overheads of the runtime profitability check, this comparison is combined with other, similar loop versioning transformations that may be performed to cope with aliasing and/or alignment conditions. In any case, the added overhead of this runtime check is taken into consideration.

2.2 Profitability equation

A general, oversimplified equation for the return value of `vect_estimate_min_profitable_iters()`, is the minimum number x that satisfies:

$$SOC + xSIC > VOC + [x/VF]VIC$$

where

`SOC` represents the total cost of instructions executed outside the loop in the scalar version,

`SIC` represents the total cost of instructions executed by one iteration of the scalar version,

`VOC` represents the total cost of instructions executed outside the loop in the vector version,

`VIC` represents the total cost of instructions executed by one iteration of the vector version.

The above equation is oversimplified for two main reasons: vectorization can apply various transformations in addition to reducing the number of iterations by a factor of `VF`, and the costs of scalar and vector operations can vary greatly between different architectures. We next explain how this equation is expanded to address these issues.

When vectorizing a loop whose number of iterations is unknown or not divisible by `VF`, an adequate number of "leftover" iterations are executed in their original scalar form after the vector loop. This number is always less than `VF`, and is taken to be `VF/2` on average.

Another case where such loop peeling takes place is when we align an address referenced in the loop to start on a preferred alignment boundary. Here again the number of peeled scalar iterations (taking place before the vector loop) is less than `VF`, and `VF/2` can be taken as average.

The expected number of scalar iterations before and/or after the vector loop, referred to as prologue and epilogue iterations, respectively, are therefore deducted from the expected number of vector iterations ($[x/VF]$ above).

Additional transformations that impact the profitability equation are related to specific techniques of vectorization, including vectorizing non-unit strided access to memory, outer-loop vectorization and intra-loop vectorization (also known as "SLP"). The associated costs are taken into consideration when performing the appropriate analysis.

2.3 Interaction with Abstract Streaming Machine (ASM)

The expected profitability of vectorizing a loop strongly depends on certain characteristics of the underlying platform, including the available mechanisms to support load from and stores to arbitrarily aligned addresses, the means to perform reduction operations, as well as the basic latencies of vector instructions compared to their scalar counterparts (where available), the costs of adding a branch (both the taken and non-taken directions), and more. The cost model interacts with the ASM of the underlying platform to acquire this information, through a set of `vect_model*_cost()` functions and `TARG_*_COST` definitions. This way, one cost model engine can support multiple platforms; this was intended by design, and achieved by active collaboration with other members of the GCC community. Each platform can, and some already have, specified such information [4].

References

1. "[patch] Vectorizer cost model implementation" – initial patch implementing the cost model, and related discussions with members of the GCC community.
(<http://gcc.gnu.org/ml/gcc-patches/2007-04/msg00193.html#01684>)
2. "RE: [patch] Vectorizer cost model implementation" – follow-up patch responding to various review comments and suggestions, including testcases, and related discussion.
(<http://gcc.gnu.org/ml/gcc-patches/2007-06/msg00361.html#00484>)
3. "[patch] [autovect] vectorizer cost-model: spu testcases and costs + fixes" - This patch adds vectorizer cost model testing for the SPU, and also defines the cost-model target-specific costs for the spu. This patch also includes several small fixes/enhancements to the cost-model itself, found during testing.
(<http://gcc.gnu.org/ml/gcc-patches/2007-06/msg01981.html#01981>)
4. "[patch] more vectorizer costmodel fixes/improvements + spu costs (needs review) " – this patch brings over to GCC4 mainline the cost model patches from the autovect development branch
(<http://gcc.gnu.org/ml/gcc-patches/2007-07/msg00496.html#00496>)
5. "[RFC][PATCH] New Vectorizer APIs" – this patch provides an initial set of API's exposing the cost model to external uses and high-level transformations, checking if a loop can be vectorized and if so what the expected profitability is.
(<http://gcc.gnu.org/ml/gcc-patches/2008-08/msg01258.html>)