RAVAC compiler User Guide

Document version 1, revision 3, 26.3.2012

CONTENTS

1 INTRODUCTION

This document describes the usage and implementation of the RAVAC (Robust Automatic Vector Accelerating Compiler) that is a compiler aimed to automatically use programmable vector accelerators.

The primary target for this compiler is the ASVP platform (Application-Specific Video Processor, originally named EdkDSP) developed by UTIA.

Since no existing and relatively recent compilation tool for a quite specific ASVP platform could be used, a new compiler had to be developed from scratch. Work on the RAVAC started at the Brno University of Technology with the goal of automatic and efficient compilation for AVSP.

Terminology

The name RAVAC was chosen quite recently. In the SMECY documents and in e-mail communication, a name BUT compiler (also times Vecta) was used. Since now (17.5.2012), RAVAC will be the only used name by BUT, however other SPECY partners may still use the term BUT compiler.

In this document, new name AVSP for the EdkDSP platform will always be used. The platform was renamed by UTIA in 2011.

2 DESIGN

2.1 *Approach to compilation*

The first considered approach for programming the ASVP platform ([1], chapter 4) was to take whole functions that should run on the workers (BCEs, Basic Computing Elements) and compile these functions to automatically use the programmable data path. This has shown to be overly complicated because of the non-general purpose nature of the ASVP workers (mainly the limited 8-bit instruction set and small program memory).

A more compiler-friendly view of the ASVP platform is based on a principle that the platform is rather a single-core processor (MicroBlaze) that uses the BCEs as vector accelerators. BCEs have a predefined set of possible vector operations that they can perform. This set of operations is based on the Worker Abstraction Layer (WAL) API provided by UTIA, where a complex operation may consist of a sequence of provided basic operations defined by this API ([1], chapter 5).

The RAVAC compiler is able to use BCEs by abstracting them to the proposed set of vector operations, e.g. vector floating-point multiplication with accumulation.

2.2 *Usage of multiple BCEs*

The second consideration was how to use multiple BCEs at once. Several approaches were considered with dataflow programming models being, by our opinion, the most suitable.

By the UTIA's requirements, the platform should be programmed using programs in the C language.

However, implementation of such tools (e.g. Kahn process networks extraction from C language) would not be feasible in reasonable time and also for practical usage, usage of other tools such as the Compaan compiler [REF], which are also still not very automatic [REF] and have a lots of unresolved issues.

The pragmatic choice we made was to start with vectorizable loops and to off-load these loops as vector operations to BCEs. Currently, one execution thread can use one BCE. In the future, more complex analyses and loops can be used in order to use multiple BCEs at once, however.

Another important consideration was how to interact with other tools in the SMECY project. In the time of the compiler design, we knew that we can expect [REF] the tool BlueBee to profile the application and the tool Par4All to find parallelizable parts and identify them by OpenMP pragmas, so we based the design on this assumption.

So the plan was to use OpenMP pragmas generated by Par4All for coarse-grained parallelization on thread level (the ASVP's MicroBlaze runs petaLinux operating system) and profiling information from BlueBee that identifies interesting parts of code that are candidates for off-loading to BCEs, where on fine grain parallelization using vector operations is used.

By this design, when an application is executed, it runs in multiple threads as specified using OpenMP pragmas. Each thread, once it encounters offloaded vector operation, switches its execution to BCE and the MicroBlaze core is free to execute another thread.

2.3 *BCE vector operations*

Before the toolchain implementation will be described, an overview of available operations that BCEs can perform is given. This description is based on the actual ASVP v 2.1 documentation [

2.4 *Considerations on MRAPI and MCAPI*

We also investigated the usage of MRAPI and MCAPI specifications.

The MRAPI is intended to multi-core resources management and describes following features: synchronization primitives, memory primitives and metadata primitives [REF]. After a thorough study of the MRAPI, we could not find much use for it. This is mainly due to the fact, that BCEs cannot run arbitrary code because of the picoBlaze's limited 8-bit architecture and small memory. MRAPI was designed for classical (mainly homogenous) multi-core architectures and the features offered by the petaLinux system are sufficient. It is possible to translate the MRAPI constructs to the posix interface, but we could not find a good use for it that would improve the performance and resource usage on the EdkDSP.

The MCAPI defines semantics for communication and synchronization between processing cores in embedded systems [REF]. This is much more interesting, because the code for ASVP can be prepared in some form of a dataflow graph than can be further mapped on the BCEs. MCAPI defines three communication types: messages, packet channels, and scalar channel. This will be further investigated. However from the non-general nature of BCEs, the tools that would generate MCAPI constructs for ASVP must be able to identify which parts of code can be executed on the BCEs and create mapping accordingly to this.

The MC and MR APIs are quite low level and it is possible to translate them into the posix interface, but we could not find a good use for it that would improve the performance and resource usage on the ASVP.

The best realistic approach we were able to come with is to create or modify a tool that uses some dataflow programming model. Computation nodes that will be mapped onto BCEs will have limited set of operations and the application is programmed by the user accordingly to these limitations. A more complex approach could identify these dataflow computation nodes automatically; however, still a large amount of work from the user would be needed in order to transform the application to have identifiable parts that can be executed on BCEs.

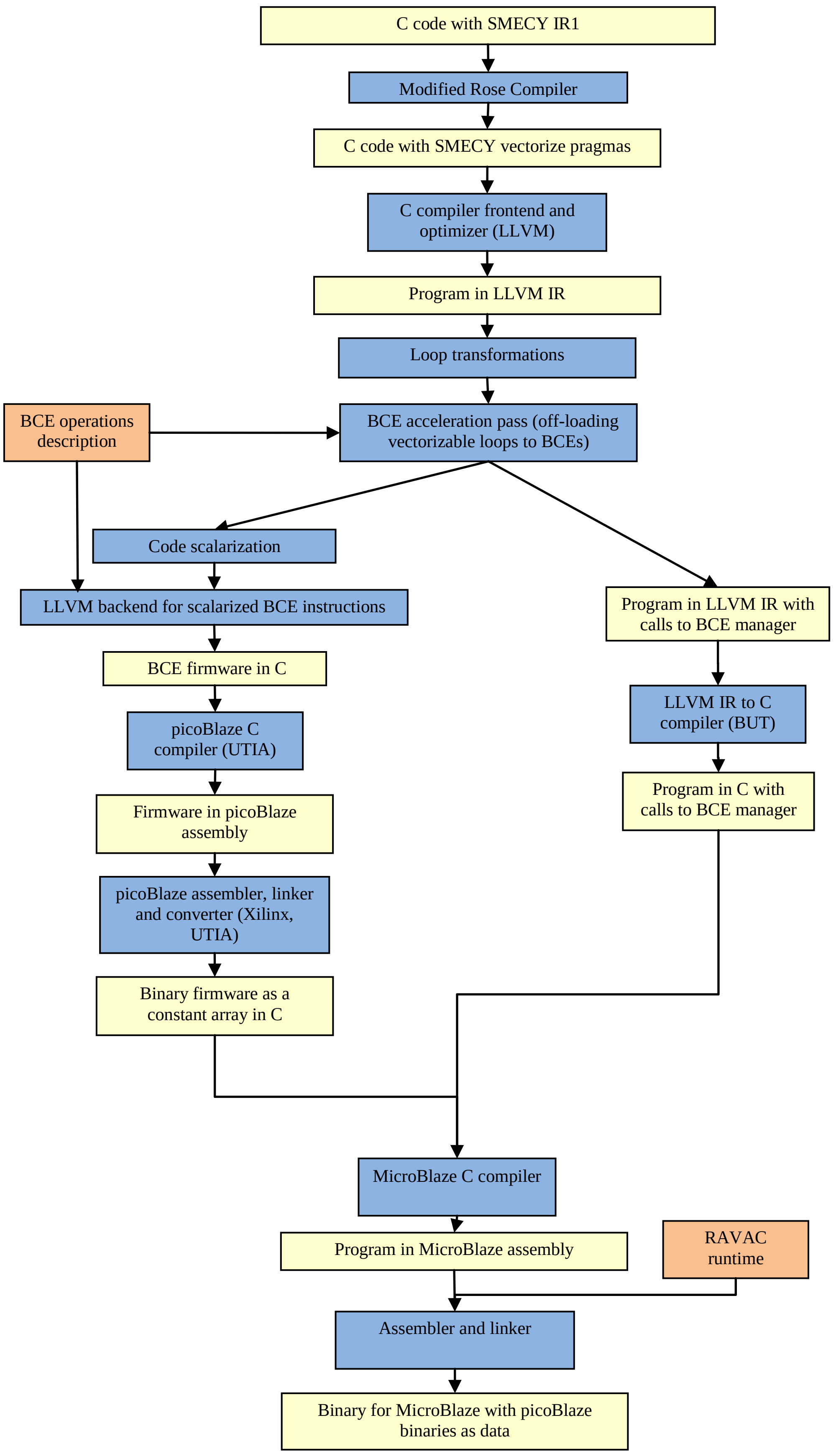
2.5 *Toolchain implementation*

As described in sections 2.1 and 2.2, the approach to compilation is based on a view that the workers have a predefined set of vector operations that they can perform in a sequence.

First, OpenMP pragmas (either generated by Par4All, or written by hand) must be replaced by pthread calls. For OpenMP replacement is the Rose Compiler [REF] used, it generates code that uses XOMP library which uses calls from GNU gomp library.

Further, vectorizable loops in code segments identified either by BlueBee or by hand are examined and if available BCE operations can be used to implement the vector operation, BCE firmware is generated and original code is replaced by calls to BCE manager. BCE manager is a layer above WAL API that takes care of dynamic firmware loading, data transfer and computation control. Vectorizable loops or pars of code that are interesting to be vectorized are marked using pragmas `#pragma smecy vectorize` (the pragma name can be easily changed). This pragma tells the compiler to off-load computations to BCE if possible.

On the following image, all the components of the RAVAC compilation chain are shown.


C code with SMECY IR1
Modified Rose Compiler
C code with SMECY vectorize pragmas
C compiler frontend and optimizer (LLVM)
Program in LLVM IR
Loop transformations
BCE operations description
BCE acceleration pass (off-loading vectorizable loops to BCEs)
Code scalarization
LLVM backend for scalarized BCE instructions
Program in LLVM IR with calls to BCE manager
BCE firmware in C
picoBlaze C compiler (UTIA)
LLVM IR to C compiler (BUT)
Program in C with calls to BCE manager
Firmware in picoBlaze assembly
picoBlaze assembler, linker and converter (Xilinx, UTIA)
Binary firmware as a constant array in C
MicroBlaze C compiler
Program in MicroBlaze assembly
RAVAC runtime
Assembler and linker
Binary for MicroBlaze with picoBlaze binaries as data

2.1 RAVAC compilation chain

Loop transformations in the Rose Compiler currently transform loops marked with #pragma smecy vectorize into loops with constant count of iterations. This is needed because the BCEs need a constant vector size. Maximal count of iterations depends on BCE local memory sizes and is configurable (currently set to 256).

Then the BCE acceleration pass analyzes loops marked with #pragma smecy vectorize. First are these marked loops analyzed and it is checked that operations used in the loop can be calculated on BCEs. Also loop iteration dependencies are checked. If all prerequisites are met, then the loop contents are extracted.

Extracted loop contents are the scalarized. This means that the original accesses to arrays are replaced with accesses to scalar variables.

Such code is then given to a modified LLVM backend whose target architecture has operations supported by the BCEs, only scalarized. Target registers represent BCE memories. Currently, there are 8 available registers A0, A1, B0, B1, C0, C1, D0, and D1. Each such register represents lower or higher half of the memories A, B, C, and D.

When the LLVM backend is run, it performs instruction selection, register allocation and instruction scheduling. This way we get a sequence of BCE operations with memory mapping for each operand. Finally, an additional pass generates BCE firmware as C code.

Now, when the firmware for BCEs is generated, we need to replace the original for loop. The original loop is analyzed, starting address and increment values for each operand are calculated and also the count of operations is determined.

The original loop is then replaced by calls to BCE manager. BCE manager is a layer above the WAL API designed to simplify the loop off-loading and firmware management.

Resulting code in LLVM IR is transformed to C, the firmwares are compiled, everything is linked together and a binary for AVSP is created.

3 USAGE

A compilation script that performs all the steps described in section 2.5 was implemented and as argument takes currently one C code file that is compiled for ASVP and outputs object files. An option to this script tells whether the loops should be off-loaded to BCEs or not.

3.1 *OpenMP pragmas*

OpenMP v. 3.0 pragmas are supported with their standard meaning. Rose Compiler and gomp libraries are used.

OpenMP pragmas are used to specify coarse-grained thread level parallelism. When part of code with off-loaded computation is executed, the thread on the Microblaze goes to sleep state and wakes up once the computation on the BCE is finished.

3.2 *SMECY Vectorize pragmas*

Currently, all for loops that can be executed on the BCE are transformed. This may not be convenient all the time and there is a need to specify which loops should be off-loaded. For this can be #pragma smecy vectorize used. It is currently supported in the modified Rose Compiler and support in the BCE acceleration pass will be added soon.

3.3 *Current limitations*

Now are all suitable for loops transformed info BCE firmware and off-loaded. The #pragma smecy vectorize is currently ignored.

Temporarily may the code inside vectorizable for loops store results into just one input array. This limitation was due to the first RAVAC version. Now, by using LLVM backend to generate firmware, it will be quite straightforward to overcome this limitation.

There must not be any inter-loop dependences in the offloaded. Backward dependences (one iteration needs result from the previous one) will not be supported in any form any time soon because of the BCE's dataflow unit pipelined nature. Currently are also forward dependences not allowed, but this will be changed soon.

3.4 *Prepared compiler in virtual machine*

For testing purposes and because of quite complicated installation of the Rose compiler, a virtual machine with 32-bit Ubuntu 12 was prepared.

The virtual machine is to be used with Virtual Box (tested with Virtual Box version 4.0.14, host machine 64-bit Fedora 15).

Username and password for the virtual machine is smecy. Root password is also smecy.

In directory /home/smecy/smecy/ravac/EdkDSPv2.0/example/ is a prepared compilation script.

Currently, there is only the LLVM part installed, so the toolchain starts from the phase "C compiler frontend and optimizer (LLVM)" in figure 2.1. If an option to use accelerators is enabled, the compiler analyzes all suitable loops in a source C program and tries to generate firmware for them.

There is also currently some Czech text present; we will translate it as soon as possible.

Script 'ravac' accepts following arguments:

```
-c   - do full compilation to a binary (will be renamed)
-u  - disable acceleration
-s SOURCE  - set input .c file as SOURCE
-l  - no loop extract (rather for debugging purposes, not needed normally)
```

A user can test the compiler by running

```
./ravac –s mac.c –c
```

A binary 'mac' for petaLinux that runs on the ASVP's and uses BCEs is then created.

Another way to compile fo AVSP is to skip the BCE usage acceleration and directly compile just for the MicroBlaze using

```
./ravac –s mac.c –c -u
```

Reference binaries for the sample mac.c program are in /home/smecy/smecy/ravac/EdkDSPv2.0/example/ref_data/.

If you encounter any problems, please contact me on ihusar@fit.vutbr.cz.

Files can be copied from and to the host machine through scp (or gftp).

4 FURTHER PLANS

A lot of work can be done in the area of loop merging and data transfer optimizations.

Special DMA hardware that e.g. copies each eight float value from the main memory to the BCEs local memory will be very useful and will improve performance especially for the multithreaded version.

5 REFERENCES

[1] SMECY, Deliverable D2.1: Preliminary report on platform dependent parameters and optimizations, v.1.0, 2.8.2010. (Available on www.smecy.eu/documents/T0+06 deliverables/D2.1/D2.1_1v0_after_revision.doc)