

Creating a GCC Back End for a VLIW-Architecture

Adrian Prantl

On Demand's Control Processor

The On Demand Control Processor is part of On Demand's scalable video engine (SVEN). The Control Processor is meant to be used as a bitstream decoder and in its basic configuration it is powerful enough to handle the parsing of H.264, VC-1 and MPEG2 streams. These are popular formats for the digital distribution of TV broadcasts and feature films.

Its main features are a 4-way VLIW core, 64 general purpose registers, conditional execution, support for various extensions and a 24 bit address space.[1]

The Control Processor is in essence a load/store architecture with a reduced instruction set (RISC) and in-order execution. Special to it are four mostly independent instruction units which are fed by very long instruction words (VLIW). These processing units are called slots. It is up to the assembler programmer and the compiler to properly distribute the single instructions across the slots.

In its standard configuration, the processor has a total of 64 general-purpose registers, each with a length of 24 bits. Two of the four slots can be provided with a condition which is evaluated by one of the other two slots.

Data memory and program memory live in separate address spaces. The CPU can address 2^{24} independent words in each address space. While one data word is 24 bits long, one instruction word is significantly longer, since it has to hold 4 instructions, including all operands.

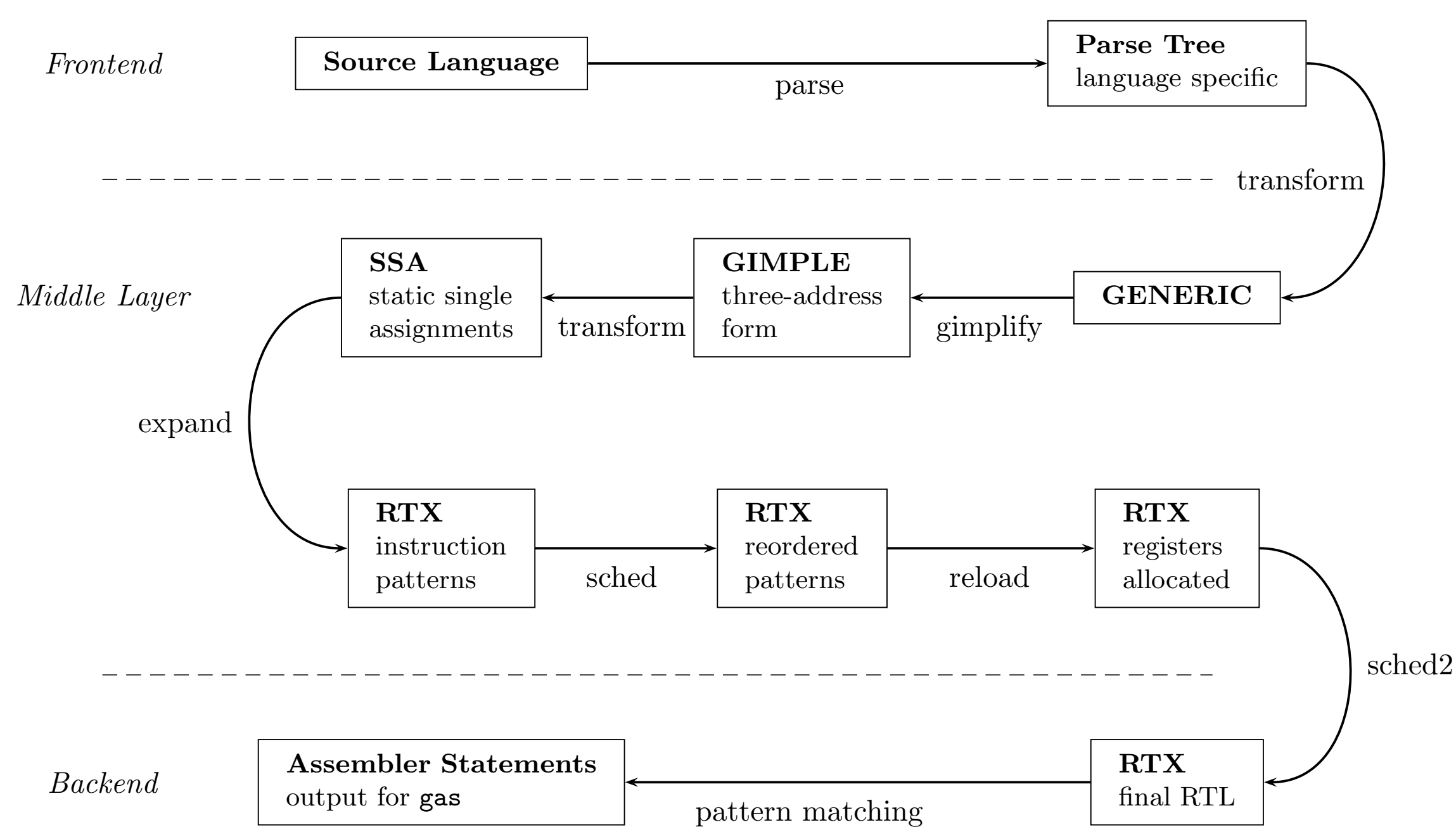
The Control Processor also supports several extensions for specialised tasks that come up when decoding modern video streams. These include algorithms like Variable Length Coding (VLC), Content Adaptive Binary Arithmetic Coding (CABAC), Content Adaptive Variable Length Coding (CAVLC), and Exponential Golomb Coding which are necessary to decode H.264 streams for example.[1]

The GNU Compiler Collection

Work on the GNU Compiler Collection (GCC) began in 1984 when Richard Stallman founded the Free Software Foundation and the GNU Project. While the GCC started as a C-compiler for the Motorola 68000 CPU it was always designed as a multi-language, multi-platform compiler. It was first released in 1987. From then the project has grown enormously and by the end of 2005 it supported over 60 host platforms, programming languages such as C, C++, Objective-C, Objective-C++, Java, Fortran and Ada and code generation for almost 40 different architectures.[2],[3]

Code generation with GCC

A compiler can be divided into three main components: The language front end which understands the source language and constructs a parse tree, a middle layer working with an intermediate representation where optimisations and other program transformations take place and a target machine specific back end handling the actual generation of assembler code.



Schematic overview of the passes and intermediate representation of compilation with GCC

The compilation of a program happens in many different phases, the majority of which are introduced by the optimiser. GCC uses many different kinds of representations for program code. In the front end, the programming language is parsed by the compiler and the program is stored in a syntax tree. This tree still carries language specific information which is gradually lost during the compilation.

Since the middle layer has to work with every supported language, a language independent representation is constructed by the front end. These trees, called GENERIC, represent an entire function in an intermediate language.

The GENERIC trees are then converted to GIMPLE trees that have expressions already broken down into a three address form, which is needed for all later program transformations. Data flow analysis brings the trees into a static single assignment (SSA) form, where each modification to a variable creates a new instance of that variable, so each variable is assigned a value exactly once.

The most important intermediate representation is the register transfer language (RTL). RTL expressions - when printed as a debug information - look very much like Lisp expressions. These lists of expressions (called RTX for *register transfer expressions*) are used for most "classic" optimisations and eventually also for the actual code generation.

Code generation works by simple pattern matching of instruction templates (which are - surprise - written in a Lisp-like form) with the register transfer lists.[4],[5],[6]

References

- [1] Siegfried Winterheller. *Control Processor Programmer's Guide*. On Demand Microelectronics, 2005.
- [2] Richard M. Stallman. The gnu project. <http://www.gnu.org/gnu/thegnuproject.html>, 2006.
- [3] The Free Software Foundation. *Using the GNU Compiler Collection (GCC)*, 2006.
- [4] Richard M. Stallman et al. Gcc internals manual. <http://gcc.gnu.org/onlinedocs/gccint/>, 2005.
- [5] Hans-Peter Nilsson. Porting gcc for dunces, 2000.
- [6] Jan Parthey. Porting the gcc-backend to a vliw-architecture. Master's thesis, Chemnitz University of Technology, 2004.

The path to the solution

A development environment consists of many components (compiler, linker, debugger and runtime library) that work closely together. To create the toolchain for the Control Processor, the following steps were completed:

- Designed a new GCC back end to generate assembler code for the On Demand Control Processor
- Extended the existing assembler for the Control Processor to generate ELF-object files (Executable and Linkable Format) with symbol tables and relocations.
- Ported the GNU Linker (ld) to work with these object files.
- Extended the GNU Debugger (gdb) to interface with the instruction-level simulator for the Control Processor.
- Added support for the Control Processor to the C-runtime library newlib.

Features of the new GCC back end

The most challenging part of the project was the GCC back end, which consists of the machine description and a lot of supporting C code. Taking account for the Control Processor's design, the following features were implemented in the back end:

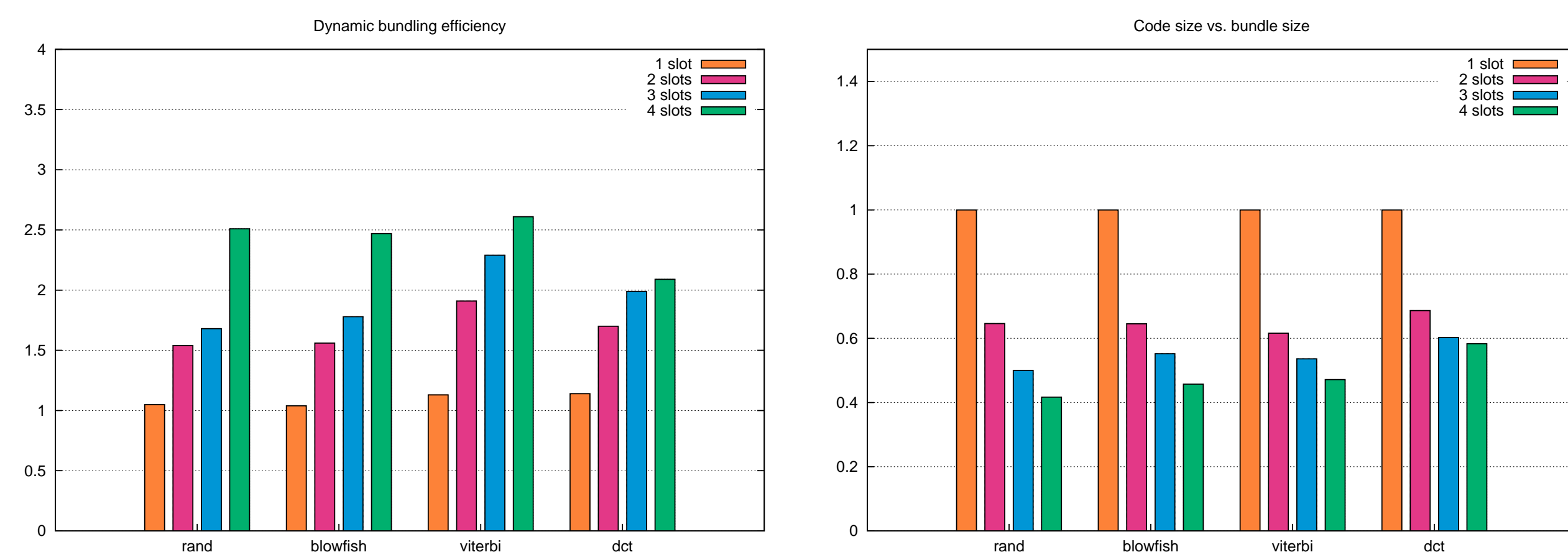
- Support for a 24-bit processor
- Pipeline model for the instruction scheduler
- Packing of VLIW bundles

The presented GCC back end uses a simple pipeline description to model the functional units of the Control Processor for the instruction scheduler. Based on the results of the scheduler, a separate pass assigns the instructions to the slots of a VLIW bundle. Using this technique an average utilisation of up to 2.5 instructions per bundle is achieved. Special care was taken to support the Control Processor's unusual byte-length of 24 bits, which affected many design decisions.

Benchmarks

The four test programs used to generate these graphs were a pseudo-random number generator (rand), a data encryption algorithm (blowfish), a method to perform error-correction on digital signals (viterbi) and a discrete cosine transform (dct).

These benchmarks show the combined effect of the instruction scheduler and the VLIW bundling phase that was implemented in the GCC back end for the Control Processor. The first graph shows how many instructions can be packed into a VLIW bundle, depending on the number of slots available. These measurements are *dynamic*, which means that they take in account how often a certain instruction word is executed. The second graph shows the reduction of the total number of instruction words with respect to the size of a VLIW-bundle. This is the *static* counterpart of the first graph.



The graphs show that the current way of instruction bundling yields about a twofold increase in performance as well a twofold reduction of the number instruction words per program.

Conclusion

The final compiler shows that GCC is fit for VLIW architectures, as it creates parallel instruction bundles that exploit the capabilities of the target processor to a high degree. The experimental evaluation also shows that some kind of global register allocation would be an advantage, since GCC has difficulties with using the many registers the Control Processor offers.

Regarding the unusual byte-length of the Control Processor, it seems that many of the GNU tools would profit from a generalisation of the assumptions they make on a CPU's characteristics. But it was also shown that it is still possible - but not necessarily easy - to create a 24-bit tools from their current versions. GCC itself does not have a problem with byte-lengths that are not a power of two as long as the target processor is still byte-addressable.

The performance of the generated code is a good start; the efficiency is increased by the parallelism the VLIW slots offer, still there are some features left waiting to be implemented, such as conditional execution and a better register allocation mechanism.