Comrade - A Compiler for Adaptive Systems

Hagen Gädke

gaedke@eis.cs.tu-bs.de

E.I.S. - Tech. Univ. of Braunschweig – Germany

http://www.eis.cs.tu-bs.de

Andreas Koch

koch@esa.informatik.tu-darmstadt.de

ESA - Tech. Univ. of Darmstadt – Germany

http://www.esa.informatik.tu-darmstadt.de

Abstract

The Comrade flow compiles ANSI-C without restrictions or annotations into combined HW-SW-solutions for reconfigurable adaptive computers, exploiting both a conventional CPU and a reconf. compute unit.

1. Compile Flow

The compiler front-end, based on the Stanford SUIF2 framework, creates a control flow graph (CFG) intermediate representation. A HW-SW partitioning pass then identifies compute-intense kernels (generally loops, Sec. 2) for possible hardware realization. This pass considers execution frequencies obtained by dynamic profiling, HW feasibility (floating point computations and library calls stay on the CPU due to excessive area requirements) and estimated speedup w.r.t. execution on the CPU. Meta information about candidate HW operations is obtained from our generic hardware module library GLACE (Sec. 3).

Each HW kernel, potentially consisting of multiple basic blocks, is then converted to a static single assignment (SSA) form, from which a control memory data flow graph (CMDFG, Sec. 4) is extracted. The CMDFG combines the pure data flow part with additional control and memory dependence information, extracted from the CFG. From each CMDFG, a controller is generated.

The controllers support dynamic scheduling for variable latency operators as well as speculative execution (Secs. 4, 5). The operation nodes of the CMDFGs are compiled into optimized pre-placed netlists, created by the GLACE generators; each controller is realized as a Verilog module.

Multiple HW kernels can be merged into a single FPGA configuration in order to reduce reconfiguration time. Each configuration is equipped with a technology-independent, configurable multi-port memory access core [1], providing cached and streaming memory accesses as well as an interface between CPU and HW kernels.

At this point, the operation of the HW kernels can be visualized by postprocessing the output of a Verilog simulation. An automatic floorplanner back-end providing a regular layout of the various components on the FPGA is already under development. It will complete the entire design flow, ranging from C to actual hardware/software co-implementations.

2. Loop Duplication

For identifying compute-intense kernels, we do not limit hardware execution to inner loops or predefined loop structures. Comrade duplicates *arbitrarily* nested loops in the CFG to analyse all reasonable HW/SW partitioning combinations of each loop nest.

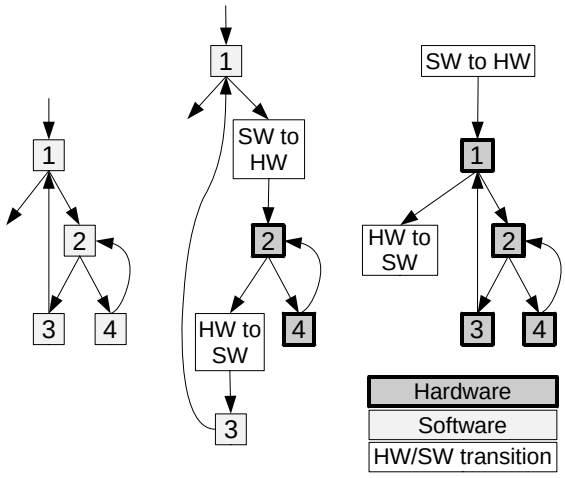

Figure 1: Partitionings for a loop nest of depth two

Fig. 1 shows the three different HW/SW partitionings from all SW to all HW for two nested loops.

3. Module Generators

The actual hardware circuits underlying the CMDFG operators are handled by the GLACE module library [2]. It provides a parametrizable generator for each basic operator that can supply the circuit appropriate for the exact operand bit-widths, data types, and throughput required. In addition to the creation of target-technology optimized macros, GLACE also offers meta-information about each operator instance (area requirements, estimated clock frequencies, latency etc.) that are used as the basis for the HW/SW trade-offs in the partitioning algorithm.

4. Control Memory Data Flow Graphs

The CMDFGs are generated from the SSA-converted CFG of the hardware candidates. A CMDFG consists of nodes for arithmetic, logic, memory access, and I/O registers (for communication with the CPU), connected by data, control and memory dependence edges.

Each SSA phi node is translated into a multiplexer in the CMDFG. The multiplexer inputs are connected to control dependence edges originating from the associated condition (Fig. 2b). I. e., not the beginning of a data flow path is controlled by a condition, but its end, which is essential for speculative computation (Sec. 5).

Memory accesses (MA), however, are not executed speculatively: a control edge is connected to each MA node. Furthermore, MA nodes are connected by memory edges to guarantee correct order of execution (e.g., WAW, WAR).

5. Fast Speculation with Down Tokens

Speculation is a well-known method for increasing computational performance. The CMDFG in Fig. 2b results from the example C code in Fig. 2a. In this example, speculative computation means that both branches of the if condition are precomputed in parallel, before the actual value of the condition `c` is known. As soon as `c` is evaluated, one of the precomputed values for `r` is passed on through the multiplexer.

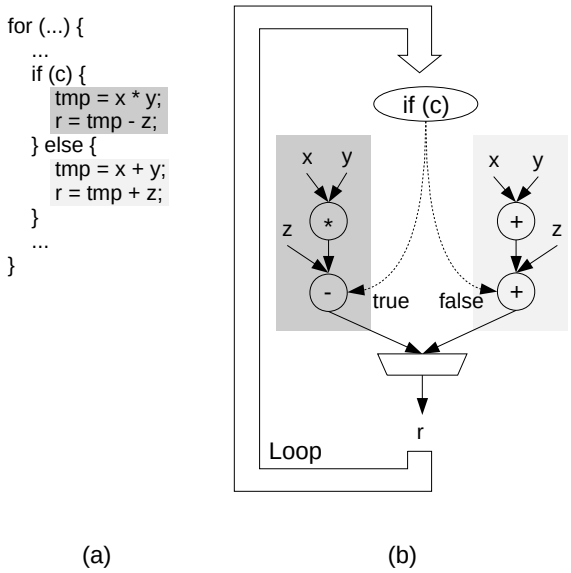

Figure 2: Example C code and resulting CMDFG

Unfortunately, when using this concept in loops, conditionals that need a different number of cycles in each branch of computation can lead to a mix-up of the precomputed results of different iterations. To overcome this issue, different approaches have been proposed.

The Pegasus [2] concept in practice requires all conditional branches to complete before the next iteration can start. FIFOs can be used to overcome this restriction, but only if all operations in the branch are pipelinable. Random memory accesses, for example, are not pipelinable.

Another approach is to use sequence tokens [3] for each precomputed value, which give a clear mapping to the associated loop iteration. The drawback is a greater amount of required space to save the sequence tokens for each basic block.

Comrade uses a different approach: computations in non-taken branches are explicitly cancelled. Example: Assume that the precomputation of both branches in Fig. 2 starts before cycle 1, while `c` is still unknown. The precomputation of the right branch is finished after 2 cycles; the left branch needs 9 cycles. Now if `c` is ready after cycle 2, the computed result of the right branch will immediately be used for consecutive computations, while the left branch of the computation for the current iteration will be cancelled. We use special tokens, called *down tokens*, to cancel operations.

For each multiplexer, a down token is assigned to each input port that corresponds to a non-taken branch. Down tokens move in the CMDFG backwards along data edges until they collide with an up token, which is the standard type of token, representing a finished computation. Colliding up and down tokens vanish. In this manner, all computations in non-taken branches are cancelled.

Using down tokens, our CMDFG implementations take advantage of speculative computation while the required additional storage is just a single down token bit for each registered operation.

6. Configuration Scheduling

In this fashion, we generate one compute unit (CU) for each of the hardware partitions. However, since we are compiling from standard C instead of a dedicated parallel programming language, the available ILP generally leads to CU sizes of less than 200 operators. This is only a fraction of the available logic capacity even of low-cost reconfigurable devices such as FPGAs. Thus, we employ this surplus of area to pack multiple CUs into each actual device configuration, reducing the time spent on reconfiguration. Based on a dynamic execution profile and the CU area requirements, we can compute the optimal packing [5].

7. Conclusion and Future Work

While additional work is required in the back-end to complete the flow down to actual hardware netlists, the tools are already able to create simulatable Verilog models of the final RCUs. These results can be used to guide further research in optimizing and refining the currently employed techniques. Among the improvements planned are a higher-performance module library as well as support for parallel memory accesses, which are already implemented in MARC [1].

8. References

[1] H. Lange, A. Koch, "*Memory Access Schemes for Configurable Processors*", Intl. Conf. on Field Programmable Logic and Applications (FPL), 2000.

[2] T. Neumann , A. Koch, "*A Generic Library for Adaptive Computing Environments*", Intl. Conf. on Field Programmable Logic and Applications (FPL), 2001.

[3] M. Budiu, S. Goldstein, "*Pegasus: An Efficient Intermediate Representation*", School of Computer Science, Carnegie Mellon Univ., 2002.

[4] H. Styles, W. Luk, "*Branch Optimization Techniques for Hardware Compilation*", Intl. Conf. on Field Programmable Logic and Applications (FPL), 2003.

[5] N. Kasprzyk, J.C. van der Veen, A. Koch, "*Configuration Merging for Adaptive Computer Applications*", Intl. Conf. on Field Programmable Logic and Applications (FPL), 2005.