COMRADE - A COMPILER FOR ADAPTIVE COMPUTING SYSTEMS USING A NOVEL FAST SPECULATION TECHNIQUE

Hagen Gädke

Integrated Circuit Design (E.I.S.)
Tech. Univ. Braunschweig, Germany
gaedke@eis.cs.tu-bs.de

Andreas Koch

Embedded Systems and Applications Group (ESA)
Tech. Univ. Darmstadt, Germany
koch@esa.informatik.tu-darmstadt.de

1. INTRODUCTION

Several examples have shown that adaptive computers are capable of outperforming traditional workstations in terms of computing time as well as energy efficiency [4] [3]. Developing applications for an adaptive computer, however, is often a complex task. Hardware (HW) and software (SW) parts as well as their interfaces have to be implemented, requiring specialized skills as well as additional design time.

To overcome this dilemma, the Comrade system compiles full ANSI-C to combined HW/SW applications for adaptive computers. Comrade is a third-generation tool, being based on concepts of predecessors GarpCC [2] and Nimble [7]. Other high level compilers have been presented, but none of them (to our knowledge) support compilation from a *full* high level input language to combined HW/SW solutions.

This paper gives an overview of Comrade's architecture and presents our current and planned future research.

2. COMPILER ARCHITECTURE

The SUIF2-based Comrade front-end creates a control flow graph (CFG) as intermediate representation. Then, a HW-SW partitioning pass identifies compute-intense kernels (generally loops) for possible hardware realization. Comrade examines all reasonable partitioning combinations for loop nests, ranging from just the innermost loops to the entire loop nest. The partitioning 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 over the CPU. Meta information about candidate HW operations is obtained from our generic hardware module library GLACE [8].

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) is extracted. A CMDFG (an example is shown in Fig. 1b) 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. SSA phi nodes correspond to multiplexers in the CMDFG. The multiplexer inputs are controlled by the associated condition. Note that not the beginning of a data flow path is controlled by a condition, but its end. This is essential for speculative computation (Sec. 3). Memory access (MA) nodes are connected by memory dependence edges to guarantee correct order of execution.

From each CMDFG, a controller is generated as Verilog module, supporting dynamic scheduling for variable latency operators as well as speculative execution. GLACE generates an optimized pre-placed circuit for each operator node, taking into account the exact operand bit-widths and data types.

Multiple HW kernels can be merged into a single FPGA configuration in order to reduce reconfiguration time [5]. Each configuration is equipped with a technology-independent, configurable multi-port memory access core [6], 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 HW/SW co-implementations.

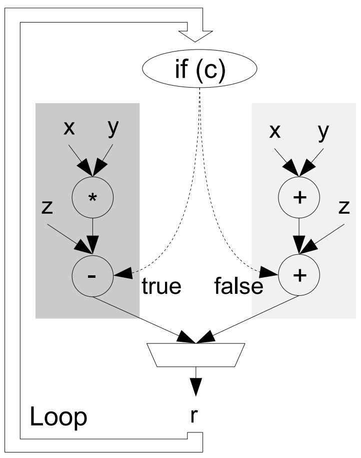
3. FAST SPECULATION WITH CANCEL TOKENS

Speculation is a well-known method for increasing computational performance. The CMDFG in Fig. 1b reflects the code example in Fig. 1a. Here, 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.

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 re-

```
for (...) {
   ...
   if (c) {
      tmp = x * y;
      r = tmp - z;
   } else {
      tmp = x + y;
      r = tmp + z;
   }
   ...
}
```

(a)



(b)

Fig. 1. Example C code and resulting CMDFG

sults of different iterations. To overcome this issue, different approaches have been proposed. Pegasus [1] in practice requires all conditional branches to complete before the next iteration can start. Another approach is to use sequence tokens [9] 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 novel approach: computations in non-taken branches are explicitly cancelled. Example: Assume that the precomputation of both branches in Fig. 1 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 *cancel tokens*, to cancel operations. For each multiplexer, a cancel token is assigned to each input port that corresponds to a non-taken branch. Cancel tokens move in the CMDFG backwards along data edges until they collide with an activate token, which is the standard type of token, representing a finished computation. Colliding activate and cancel tokens vanish. In this manner, all computations in non-taken branches are cancelled.

Using cancel tokens, our CMDFG implementations take advantage of speculative computation while the required storage is just a single additional cancel token bit per CMDFG edge.

4. PRELIMINARY RESULTS AND FUTURE WORK

Table 1 shows benchmarks of two kernels: the "parallel" kernel, which computes 50 additions per loop iteration, and the adpcm coder (audio compression) kernel. The parallel kernel shows the impact of parallelism on the kernel runtime: the HW kernel is faster than a Pentium M which has 20 times the kernel clock frequency on the FPGA. For many typical applications, however, finding sufficient ILP is more involved. Kernels containing memory accesses (like adpcm) are currently not executed speculatively. Here, the Pentium still is an order of magnitude faster than the HW.

Table 1. Kernel benchmarks for the parallel kernel (100 iterations) and the adpcm kernel (100 input samples)

	parallel	adpcm
Pentium M @ 2.13 GHz	8.1 μs	8.4 μs
HW kernel @ 100 MHz (sim.)	5.6 μs	46.2 μs

Based on our novel speculative controller model, which enables us to explore more optimization strategies than before, we are going to examine the following approaches in our research: parallel and multi-threaded memory accesses, memory localization, speculative loads, use of dedicated HW ressources (such as MAC units on recent Virtex devices) and further improvements of the HW module generators (exploiting bit granularity constants and supporting floating point operators). Another important objective is the completion of the HW backend, enabling us to actually test the full HW/SW co-implementation flow.

5. REFERENCES

[1] M. Budiu, S. Goldstein, "Pegasus: An efficient intermediate representation", Technical Report, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, USA, Apr. 2002.

[2] T. Callahan, J. Hauser, J. Wawrzynek, "The Garp architecture and C Compiler", IEEE Computer vol. 33(4):62–69, 2000.

[3] O. D. Fidanci et al., "Performance and Overhead in a Hybrid Reconfigurable Computer", Proc. 17th IEEE Intl. Parallel and Distributed Processing Symposium, Apr. 2003.

[4] H. Gädke, A. Koch, "Wavelet-based Image Compression on the Reconfigurable Computer ACE-V", Proc. 14th Intl. FPL Conf., Aug. 2004.

[5] N. Kasprzyk, J. van der Veen, A. Koch, "Configuration Merging for Adaptive Computer Applications", Proc. 15th Intl. FPL Conf., pp. 217–222, Aug. 2005.

[6] H. Lange, A. Koch., "Memory Access Schemes for Configurable Processors", Proc. 10th Intl. FPL Conf., Aug. 2000.

[7] D. MacMillen, "Nimble Compiler Environment for Agile Hardware", Storming Media LLC (USA), 2001.

[8] T. Neumann, A. Koch, "A Generic Library for Adaptive Computing Environments", Proc. 11th Intl. FPL Conf., pp. 503–512, Aug. 2001.

[9] H. Styles, W. Luk., "Branch Optimization Techniques for Hardware Compilation", Proc. 13th Intl. FPL Conf., Aug. 2003.