Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens*

Hagen Gädke[1] and Andreas Koch[2]

[1] Integrated Circuit Design (E.I.S.)
Technische Universität Braunschweig, Germany,
gaedke@eis.cs.tu-bs.de
[2] Embedded Systems and Applications Group (ESA)
Technische Universität Darmstadt, Germany
koch@esa.informatik.tu-darmstadt.de

Abstract. We present an improved method for scheduling speculative data paths which relies on cancel tokens to undo computations in misspeculated paths. Performancewise, this method is considerably faster than lenient execution, and faster than any other known approach applicable for general (including non-pipelined) computation structures. We present experimental evidence obtained by implementing our method as part of the high-level language hardware/software compiler COMRADE.

1 Introduction

Many compilers for compilation from C to hardware have been developed in the last few years; the following list is far from complete: GarpCC [1], NIMBLE [2], CASH [3], SPARK [4], ROCCC [5], Tartan [6], COMRADE [7, 8]. All of these compilers make use of speculative execution to produce efficient hardware realisations of sequential code. Speculative execution in hardware means the technique of computation without knowing if such precomputed data will actually be needed for successive computations. Applying this concept to if statements results in computing both, the then and the else block in parallel, as soon as data dependencies are fulfilled. Analogously, all cases of a switch statement would also be computed in parallel. In the literature, this approach is referred to as an upward code motion before the condition [9], or a weakening of guards, in the extreme case: the complete removal of guards or replacement of all predicates by **true** [10]. Such speculative data paths are combined in multiplexers. Only one of the computed values, depending on the result of the control condition or predicate, flows through the mux to subsequent data paths, the others are discarded. In many cases these parallel, speculative data paths have different lengths, resulting in different computation time periods or numbers of cycles. Control conditions, often being composed of simple comparisons, are typically computed much faster than the longest speculative data path. Assuming that the result of the short data path in Fig. 1 as well as the control condition cond

*Revised version, 080319

are already available, and cond is true, then the value of x2 will pass the mux, while the current computation of the else block must be discarded to prevent erroneous results.

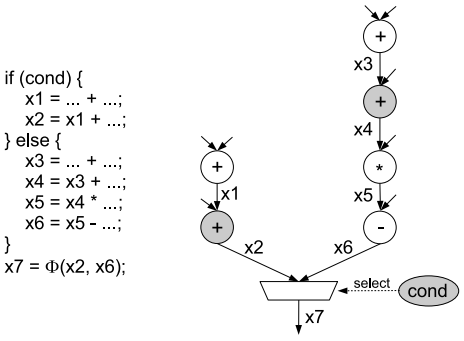

Fig. 1. Speculative data paths of different length. Gray highlighted operations denote currently available results.

Section 2 outlines related work and motivates our approach of using *cancel tokens*. After describing the intermediate representation that we use to implement cancel tokens (Sec. 3), we present details on their functionality (Sec. 4) and give a solution for the problem of control redundance (Sec. 5), which is inherent in cancel tokens. Section 6 summarises the advantages of cancel tokens, before we conclude with experimental results.

2 Related Work

The trivial approach for discarding speculated computations is to wait until *all* speculative data paths have finished their computation and only then forwarding the valid result through the mux.

The problem of waiting for the critical speculative data path is irrelevant if the traditional approach of *hardware loop pipelining* is used (such as in the StReAm compiler [11]). Short data paths can simply be extended with flipflops to match the length of critical paths. As long as the pipeline is efficiently used, computing multiple loop iterations at the same time, the length of the pipeline itself does not affect the throughput. However, if the loop contains a loop-carried dependency (LCD), the associated computation for iteration $n + 1$ cannot start before the value from iteration n has been computed. In such a situation, a pipelined solution suffers from the same latencies as the trivial approach.

Lenient execution is used by Pegasus/CASH [3]. This method allows an operator to compute its result and forward it to successive operators, although not all data inputs are available yet. Typical examples are lenient ANDs, ORs and

multiplexers. Note that lenient execution does not cancel any mis-speculated inputs.

Styles and Luk [12] present a method to accelerate the execution of loop nests, which contain an inner loop with LCDs while the outer loop is a non-LCD loop. During cycles in which the pipeline inputs of the inner loop are stalled (waiting for the LCD data to appear), an new iteration of the outer loop is executed in an overlapped fashion. Sequencing tokens are attached to the data items generated in the outer loop, identifying the loop iteration. Using these sequencing tokens later allows the correct ordering of data commits.

Our approach, which we briefly outlined already in [8], is even able to accelerate LCD loops *without* the presence of an enclosing non-LCD outer loop. We explicitly cancel mis-speculated operators using cancel tokens. In Fig. 1, `x2` would then flow through the multiplexer in the next cycle, while a cancel token would be created at the node which computes `x6`. The cancel token would then move backwards along incoming data edges and finally cancel the mis-speculated computations. Thus, the runtime delays of the trivial approach are eliminated, while mis-speculated results are deleted. This methodology is not only able to accelerate LCD loops — it performs very well in designs employing non-pipelined operators, too: Cancelling a non-pipelined high-latency operator for a mis-speculation in the current iteration allows this operator to start the computation for the next loop iteration earlier. Furthermore, if speculative, cached memory accesses are used (something we intend to tackle next), cancel tokens can remove mis-speculated accesses from the load/store queue before they are actually executed. This increases the cache efficiency, both by reducing the general demand on cache bandwidth (fewer loads/stores in general) as well as limiting the cache thrashing (fewer mistakenly evicted lines due to mis-speculated loads).

A similar methodology for killing selected computations before they are completed has first been presented by Brej [13] (later extended by Ampalam [14], who correctly addresses metastability issues), but at the lower level of asynchronous gates in ASIC designs. Our own, independently developed approach for performing such actions on the higher-level of synchronous operators was first introduced in [7] and explained in more detail in [15]. The novel techniques we present in this work refine that scheme to handle cancellation of nested conditional constructs, which requires the construction of an efficient forwarding mechanism for cancel tokens along control edges.

3 CMDFGs

Before the functionality of cancel tokens is explained (Sec. 4), we describe the *CMDFG* (Control Memory Data Flow Graph), the intermediate representation (IR) we use to support cancel tokens in the COMRADE compiler.

Fig. 2(a) shows a C code sample which will be referred to as *test* throughout the paper. Fig. 2(b) shows the corresponding control flow graph (CFG) in static single assignment-form (SSA), Fig. 2(c) depicts a section of the resulting CMDFG. The CMDFG is a low-level, fine-grain IR similar to the program depen-

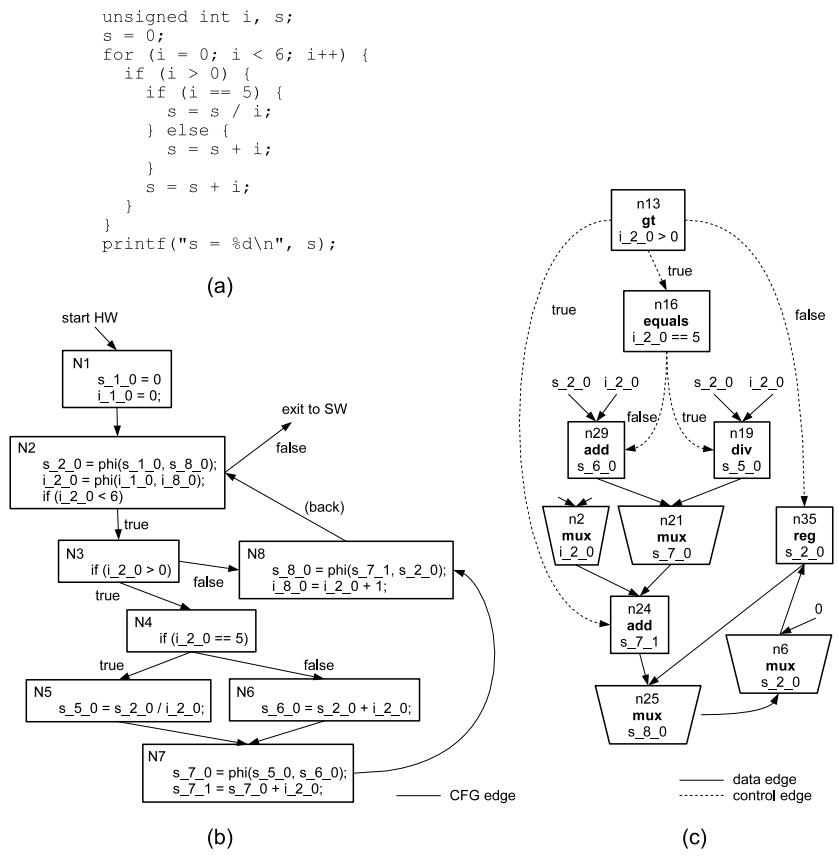

Fig. 2. *test* — (a) sample C source code; (b) SSA form CFG; (c) CMDFG for the loop body (redundancy not yet removed).

dence graph (PDG) [16]. Its nodes are HW operations (arithmetic, logic, mux, registers, I/O), connected by three different types of edges that represent data, control and memory dependencies. The latter, being out of scope of this paper, guarantee the correct execution order of memory access nodes. Without memory edges, the CMDFG is somewhat similar to the dependence graph as used in [10] as well as the program dependence web (PDW) [17]. However, CMDFG nodes do not contain complete data flow graphs like in this prior work, but single HW operators (similar to Pegasus [3] and HCDG [9]). Compared to the PDW, the CMDFG contains neither γ and β functions, nor data flow switches. Instead it employs a more hardware-centric view by relying on multiplexers, whose predecessor nodes (and, for loop-carried values: successor nodes) are targets of control edges. Note that a muliplexer's one-hot select wires are connected to the appropriate activate token registers (cf. Sec. 4) of the mux' data predecessor nodes.

Thus, a mux implicitly forwards the output value of the currently active predecessor. Sec. 4 describes how mis-speculated and superfluous values can be cancelled in CMDFGs.

4 Activate and Cancel Tokens

CMDFGs operate in the data flow paradigm: as soon as all data predecessor nodes of a node provide a datum, the node starts its computation (i.e., CMDFGs use a self-timed scheduling). For a mux, this condition is somewhat altered in that the mux computes (i.e. passes on) the datum already if the *selected* datum is available, the others are not required to be present. For any computing node that is also the target of a control edge, its result is not considered valid until the associated control condition is *also* fulfilled. A valid result of a CMDFG node corresponds to a so-called *activate token*, depicted as '+' in Fig. 3. An activated node holds one activate token per outgoing edge, the tokens each moving along their edges to the successor nodes. Conversely, cancel tokens usually (there are exceptions, see Section 5) move *backwards*, erasing data incoming from edges determined to have been mis-speculated.

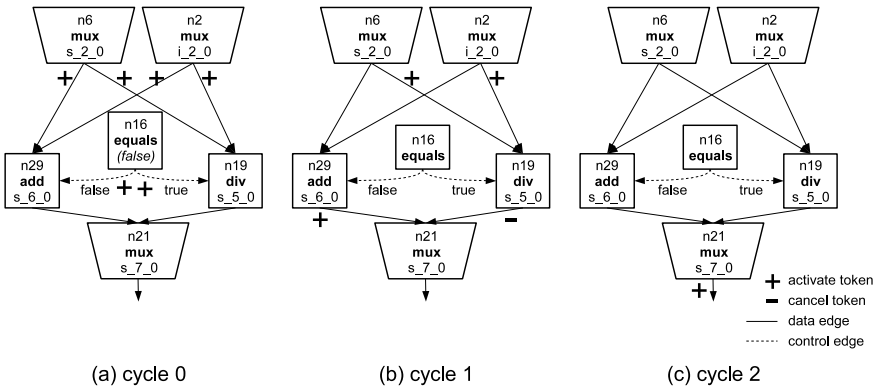

Fig. 3. Activate and cancel token flow.

In Fig. 3(a), both the activate token associated to the edge (n6, n29) of node n6 and the one associated to edge (n2, n29) of node n2 move forward along the data edges to n29, which means that n29 holds a valid result at its data output in cycle 1 (Fig. 3(b)). The figure also shows how mux predecessors are controlled via control edges. Node n16 is false in cycle 0, i.e. the left speculative data path is valid; thus, the value computed by n29 is valid in cycle 1. At the same time, a cancel token is created in n19, because it is the target of a control edge, whose control condition is *not* fulfilled. By causing these two actions, n16's two activate

tokens are consumed in cycle 1. While the valid result of the adder moves *forward* to the mux in cycle 2, the cancel token moves *backwards* along n19's incoming data edges, killing the two activate tokens remaining in the mis-speculated path.

5 Removing Control Redundance and the Control Redundance Frontier

Similar to [10], we connect nested conditionals via control edges, e.g. edge (n13, n16) in Fig. 2(c). This is reasonable in context of cancel tokens: If an outer condition has already established that the entire subgraph is not to be executed, that subgraph can be *completely* cancelled. This is achieved by making cancel tokens move *forward* along control edges. Thus, a cancel token in a control node higher up in the hierarchy is propagated down to all sub-control nodes, which then cancel all the subgraph computations.

With this approach, however, a difficulty becomes apparent: In the initial model described so far, too many cancel tokens are generated at outer levels of the control hierarchy. For example, if $i\_2\_0 \not> 0$ in Fig. 2(c), a cancel token created in n16 by n13 to disable inner conditionals cancels n29 and n19. However, the outer cancel token sent to n24, now no longer *has* a result to delete: While n2 *does* supply data (and an activate token) to the left input of n24, no data (and activate token) is forthcoming from n21, since both of that mux' inputs have already been cancelled by the inner condition. Without a datum to neutralise, the cancel token itself remains here indefinitely.

In this situation, we term n24 to be *redundantly controlled*. In order to avoid such a creation of excess cancel tokens, which do not have a corresponding activate token to neutralise, we have to remove such redundancies. One solution would be the removal of the control edge (n13, n24) and the addition of an edge (n13, n2), which would lead to the neutralisation of the activate token coming into n24 from n2.

The algorithm that can solve this problem in the general case requires some definitions.

Let T be a CMDFG with a set of nodes N, a set of data edges E_d, a set of control edges E_c, $N_m \subset N$ the set of multiplexer nodes and $N_c \subset N$ the set of *condition nodes* (i.e. nodes having outgoing control edges). The function m maps a condition node to its associated multiplexer; a multiplexer belongs to the condition which controls all of the multiplexer's data predecessors:

$$m : N_c \longrightarrow N_m, \quad m(c) := x \in N_m : \forall (z, x) \in E_d : (c, z) \in E_c$$

The function P_d maps two nodes $a, b \in N$ to the set of nodes belonging to any path along data edges from a to b:

$$\begin{aligned} &P_d : N \times N \longrightarrow \mathcal{P}(N) \\ &P_d(a, b) := \{a\} \cup \{b\} \cup \{z \in N \mid \exists\, path\ (a, \ldots, z)\ along\ data\ edges \\ &\qquad\qquad\qquad\qquad \wedge\ \exists\, path\ (z, \ldots, b)\ along\ data\ edges\} \end{aligned}$$

For the following definitions, let $S = (c_1, \ldots, c_n)$ be a *condition nest* in T, i.e. S is a path of condition nodes in T. S_i is the i-th node in S, such that

$$S_1 = c_1,\ \ldots,\ S_n = c_n;$$

n is the length of the path.

We now define the set $C(S_i)$ of nodes which are directly or indirectly *controlled by* S_i:

$$\begin{aligned} i = n: \quad C(S_i) &:= \{a \in N \mid (S_i, a) \in E_c\} && (1)\\ &\cup\ \{m(S_i)\} && (2)\\ \\ i < n: \quad C(S_i) &:= \{a \in N \mid (S_i, a) \in E_c\} && (3)\\ &\cup\ \{m(S_i)\} && (4)\\ &\cup\ P_d(m(S_{i+1}), m(S_i)) && (5)\\ &\cup\ C(S_{i+1}) && (6) \end{aligned}$$

Lines (1) and (3) add direct control successors of S_i to C, (2) and (4) add the mux belonging to S_i, (5) adds nodes of paths from a direct sub-condition of S_i (located at the deeper nesting level $i+1$) to the mux at the current nesting level of S_i, and (6) recursively adds the nodes controlled by nested conditions.

We define the set $R_I(S_i) \subset C(S_i)$ of *immediate redundantly controlled* nodes of S_i, which are directly controlled by S_i, but which would be assigned excess cancel tokens (as some of their data predecessors are already controlled by S_i):

$$R_I(S_i) := \{a \in C(S_i) \mid \exists (S_i, a) \in E_c\ \wedge \exists (z, a) \in E_d : z \in C(S_i)\} \qquad (7)$$

The set $R(S_i)$ of *redundantly controlled* nodes of S_i (defined for $i < n$) now extends R_I by all paths from the sub-condition's multiplexer to immediate redundantly controlled nodes:

$$R(S_i) := R_I(S_i)\ \cup \bigcup_{a \in R_I(S_i)} P(m(S_{i+1}), a) \qquad (8)$$

Thus, $R(S_i)$ is the set of nodes that are affected by excess cancel tokens.

Finally, the *control redundance frontier* $F(S_i)$ (defined for $i < n$) is the set of nodes which are *not* controlled by S_i, but which are direct data predecessors of a node that *is* redundantly controlled by S_i:

$$F(S_i) := \{a \in N \mid a \notin C(S_i)\ \wedge \exists (a, z) \in E_d : z \in R(S_i)\} \qquad (9)$$

For example, in Fig. 4, with $S = \{n_1, n_2\}$,

$$\begin{aligned} m(n_1) &= n_{11} && m(n_2) = n_5\\ P(n_5, n_{11}) &= \{n_5, n_6, n_8, n_{10}, n_{11}\}\\ C(n_1) &= \{n_2, n_3, n_4, n_5, n_6, n_8, n_{10}, n_{11}, n_{12}\} && C(n_2) = \{n_3, n_4, n_5\}\\ R(n_1) &= \{n_5, n_6, n_8, n_{10}\}\\ F(n_1) &= \{n_7, n_9\} \end{aligned}$$

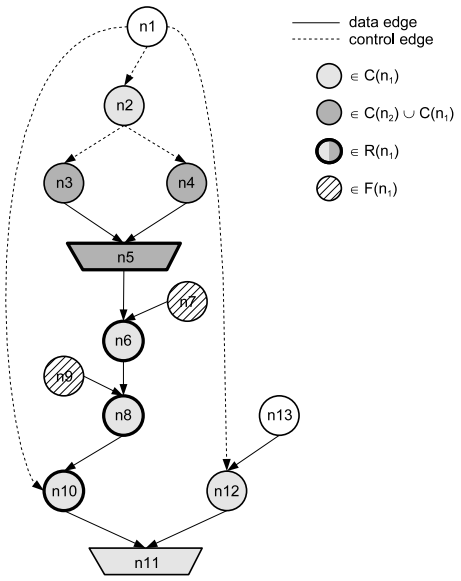

Fig. 4. Example illustrating the definitions in Sec. 5.

To avoid token imbalances, we just have to remove all redundant control edges (S_i, x) with $x \in R(S_i)$, and add instead non-redundant control edges (S_i, z) for $z \in F(S_i)$. This is precisely the solution we used at the beginning of this section.

6 Advantages of Cancel Tokens

Cancel tokens avoid delays due to waiting for the results of mis-speculated branches of the computation. However, this applies to non-pipelined computations. In pipelined computations, it is not possible to cancel a branch (sub-pipeline), since this would alter the latency of this computation path. That is of course not acceptable in a pipelined compute unit where merging paths must all be balanced to have the same latency from the input nodes. Here, mis-speculated results are always fully computed, just not used afterwards.

However, since we are aiming for the compilation of general C code, possibly with irregular control flow in loops and loop-carried dependencies (as shown for the variable `s` in Fig. 2(a)), we cannot always generate strictly pipelined compute units (the next set of input data cannot enter the pipeline before the previous result has been computed by the pipeline). In order to handle even the non-pipelined case efficiently, we rely on the cancel mechanism.

Another advantage of cancelling non-pipelined multi-cycle operators (e.g. div, transcendental functions, etc.) is that the next computation — typically corresponding to the next loop iteration — of such operators can start earlier,

reducing the execution time for the correlated loop iterations. The advantages to cancelling mis-speculated memory operations were already pointed out in Sec. 2.

7 Experimental Results

To evaluate the practical impact of our approach, we extended the COMRADE compiler appropriately. COMRADE creates combined HW/SW solutions for *adaptive computing systems* (consisting of both a software-programmable processor and a reconfigurable compute unit, [18]). After HW/SW partitioning the input C program, COMRADE transforms the HW-suitable pieces of the code to SSA form and creates a CMDFG for each HW kernel. For each CMDFG operator node, our module generator library GLACE [19] creates an optimised, pre-placed netlist. Furthermore, a central controller is generated as a Verilog netlist. It holds the token registers for each HW operator (an activate and a cancel bit per outgoing edge) and dynamically controls the data flow.

This initial implementation of the controller synthesis is not yet optimal: A target node of a control edge starts its computation only *after* the control condition has been computed; a better design would be to start computation as soon as all data dependencies are fulfilled (cf. Sec. 4). Also, all operators are currently registered, i.e. we have not exploited chaining yet. However, despite these deficiencies, we can already measure the impact of the cancel mechanism.

While we can demonstrate the actual system-level execution of COMRADE-generated HW/SW applications on a Xilinx Virtex-II Pro-based hardware platform, our techniques are applicable to all fined-grained reconfigurable devices. For the experiments, we consider HW kernels from different benchmark suites (mostly from MediaBench and MiBench), as well as three additional synthetic HW kernels of our own design: the *parallel* kernel computes 100 iterations of 50 independent additions contained in the body of an if statement; *test* corresponds to the code in Fig. 2(a); *test_unrolled* refers to the same code with the loop unrolled. After using COMRADE for compilation to HW, we obtained the values listed in Table 1 by post-place&route analysis. For checking the feasibility of the approach in actual hardware, the most demanding (in terms of token amounts) parallel kernel was also successfully executed on a prototype HW platform, achieving the 100 MHz clock frequency required for this reference design [20].

The adpcm, des and pegwit kernels contain memory loads and stores. These are currently executed non-speculatively and serialised, because COMRADE does not exploit its memory dependence analyses during hardware generation yet. While not critically relevant in context of this paper (we have not discussed the memory-dependence handling parts of the CMDFG IR here), we give some performance numbers for reference: Accesses to the DDR-DRAM-based main memory are routed through a fully-associative 4 KB cache running at 100 MHz, provided by the configurable MARC memory system [21]. For a hit, the read/write latency is 1 cycle; a miss takes 45 cycles to load a cache line of 128

	#lines	#cycles	DOP max	#AT max	#CT max	#Op
adpcm	79	4497	4	101	7	123
bitcount	5	99	2	12	1	12
des	7	437	6	57	1	74
pegwit	9	797	4	70	6	96
parallel	55	398	50	256	50	315
test	11	77	2	21	5	21
test_unrolled	57	64	7	77	12	60

Table 1. Benchmarks obtained from simulating selected kernels; #lines: number of C source code lines (HW relevant part only), excluding comments, declarations, constant array definitions and white space; #cycles: HW computation cycles only, without SW/HW transfer of live variables; DOP (degree of parallelism): max. number of parallel computing operations; #AT: max. number of activate tokens; #CT: max. number of cancel tokens; #Op: total number of HW operators.

bytes. Also, all HW operators are currently non-pipelined (a new GLACE version currently under development will change this, too).

The DOP value of the parallel kernel in Table 1 reveals that all data-independent computations are actually done in parallel. The number of activate tokens (AT) in flight exceeds the operator parallelism, since each operator stores an AT per *outgoing* CMDFG edge. Thus, an increase of operator dependencies directly translates into an increase of ATs. The number of cancel tokens (CT) is much smaller, which means that only a relatively small number of operations had to be cancelled due to mis-speculation.

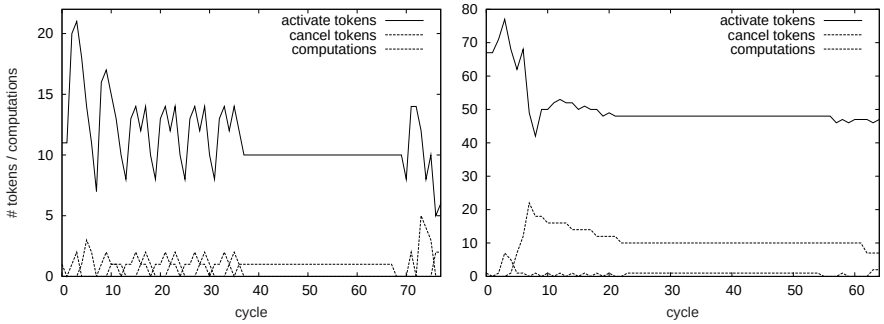

Fig. 5. *test* (left) and *test_unrolled* (right): tokens and parallel computations over time.

The left graph of Fig. 5 shows the AT, CT and DOP values for test over time. The peaks are correlated to new loop iterations. The first five iterations (6 or 9 cycles each) are much faster than the last iteration (40 cycles), because the

data path containing the high-latency divider was cancelled in all but the last iteration, demonstrating the efficiency of cancel tokens.

As the test example contains a loop-carried dependency (variable s), unrolling the for-loop achieves only a relatively small performance gain (64 instead of 77 cycles), at the large expense of tripling the number of HW operators. This shows a typical situation where pipelining would be similarly inefficient. Here, it would be more efficient to omit the unrolling and execute the loop in a non-pipelined fashion, relying instead on our cancel mechanism for speed-ups.

For the test example, Table 2 compares our cancel token approach to the trivial approach and lenient execution. The values shown are actual measurements for the cancel-token approach using non-pipelined operators. The trivial approach and lenient execution values had to be manually calculated, since no corresponding compiler implementations are available. To calculate these numbers, we assume the same operators and data flow as used by the cancel token approach.

Iteration	trivial approach	lenient execution (non-pipelined div)	lenient execution (pipelined div)	cancel tokens
	#cycles (calc.)	#cycles (calc.)	#cycles (calc.)	#cycles (meas.)
$i = 0$	40	6	6	6
$i = 1$	40	36	36	9
$i = 2$	40	36	6	6
$i = 3$	40	36	35	6
$i = 4$	40	36	6	6
$i = 5$	40	72	41	40
Total	240	222	130	73

Table 2. Measured and calculated execution times for the separate iterations of the test example. The number of cycles per iteration corresponds to the number of cycles needed to compute a new value for variable s_2_0 (see Fig. 2(b,c)).

Assuming the trivial approach, *each* iteration would require 40 cycles since the merge points in the data path would *always* have to wait for the slowest branch (the divider). Because of the LCD, this holds for both using a pipelined or a non-pipelined divider.

Lenient execution with a non-pipelined divider computes the first iteration in only 6 cycles, but consecutive iterations have to wait for the divider, resulting in 36 cycles per iteration. In the last iteration, the divider first has to finish its computation for $i = 4$ and after that additionally computes the value for $i = 5$, which results in 72 cycles.

A pipelined divider significantly increases the performance of the lenient execution version. But there are still long delays every two iterations arising from the fact that the multiplexer for s_7_0 (the successor of the divider in the data flow) cannot forward a value before the divider has completed its computation

of the previous iteration (i.e., the multiplexer cannot store the information that two or more values from one of its inputs have to be discarded).

In the cancel token approach, the divider being non-pipelined, the time-consuming data flow branch containing the divider is cancelled in the first five iterations, thus its latency affects only the iteration for $i = 5$. Here, a pipelined divider wouldn't increase performance due to the LCD.

In summary, Table 2 shows that cancel tokens can achieve significant improvements of runtime compared to the other approaches. These gains are due to the differing lengths of data path branches, which of course vary between different applications. Note that we did not yet consider the impact of reduced memory traffic due to cancelling of mis-speculated memory accesses. This will be evaluated in future work.

8 Conclusion and Future Work

We have presented a method for dynamically scheduling speculative data paths using cancel tokens, which allow the explicit cancelling of mis-speculated paths at run-time. We have already implemented cancel tokens in the hardware generation passes of the compiler COMRADE, and have thus practically demonstrated their feasibility and measured their effect on several benchmark kernels. Our method leads to significantly faster hardware than the trivial and lenient execution approaches commonly used today. The impact of cancelling in-flight memory accesses has not even been considered here yet. Future work will concentrate on implementing the mechanisms (both in the compiler as well as in the hardware memory system) allowing this cancelling of mis-speculated memory operations.

References

1. Callahan, T., Hauser, J., Wawrzynek, J.: The Garp architecture and C Compiler. IEEE Computer, Vol. 33(4), pp. 62-69, April 2000
2. MacMillen, D.: Nimble Compiler Environment for Agile Hardware. Storming Media LLC, USA, 2001
3. Budiu, M.: Spatial Computation. Ph.D. Thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA, December 2003
4. Gupta, S. et al.: SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. Intl. Conf. on VLSI Design (VLSI), New Delhi, India, January 2003
5. Guo, Z., Buyukkurt, B., Najjar, W., Vissers, K.: Optimized Generation of Datapath from C Codes for FPGAs. Intl. Conf. on Design, Automation, and Test in Europe (DATE), Munich, Germany, March 2005
6. Mishra, M., Callahan, T., Chelcea, T., Venkataramani, G., Budiu, M., Goldstein, S.: Tartan: Evaluating Spatial Computation for Whole Program Execution. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, California, USA, October 2006

7. Koch, A., Kasprzyk, N.: High-Level-Language Compilation for Reconfigurable Computers. Intl. Conf. on Reconfigurable Communication-centric SoCs (ReCoSoC), Montpellier, France, June 2005
8. Gädke, H., Koch, A.: COMRADE: A Compiler for Adaptive Computing Systems Using a Novel Fast Speculation Technique. Intl. Conf. on Field Programmable Logic and Applications (FPL), Amsterdam, Netherlands, August 2007
9. Kountouris, A., Wolinski, C.: Efficient Scheduling of Conditional Behaviors for High-Level Synthesis. ACM Transactions on Design Automation of Electronic Systems (TODAES), Vol. 7(3), pp. 380-412, July 2002
10. Gong, W., Wang, G., Kastner, R.: A High Performance Application Representation for Reconfigurable Systems. Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA), Las Vegas, NEV, USA, June 2004
11. Mencer, 0., Hubert, H., Morf, M., Flynn, M.: StReAm: Object-Oriented Programming of Stream Architectures using PAM-Blox. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa Valley, CA, USA, April 2000
12. Styles, H., Luk, W.: Pipelining Designs with Loop-Carried Dependencies. IEEE Intl. Conf. on Field-Programmable Technology (FPT), Brisbane, Australia, December 2004
13. Brej, C., Garside, J.: Early Output Logic using Anti-Tokens. Intl. Workshop on Logic Synthesis (IWLS), Laguna Beach, CA, USA, Mai 2003
14. Ampalam, M., Singh, M.: Counterflow Pipelining: Architectural Support for Preemption in Asynchronous Systems using Anti-Tokens. Intl. Conf. on Computer Aided Design (ICCAD), San Jose, CA, USA, November 2006
15. Kasprzyk, N.: COMRADE - Ein Hochsprachen-Compiler für Adaptive Computersysteme. Ph.D. Thesis, Integrated Circuit Design (E.I.S.), Tech. Univ. Braunschweig, Germany, June 2005
16. Ferrante, J.: The Program Dependence Graph and Its Use in Optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 9(3), pp. 319-349, July 1987
17. Campbell, P., Krishna, K., Ballance, R.: Refining and Defining the Program Dependence Web. Technical Report TR 93-6, Department of Computer Science, University of New Mexico, Albuquerque, NM, USA, March 1993
18. Koch, A.: Advances in Adaptive Computer Technology. Habilitation, Integrated Circuit Design (E.I.S.), Tech. Univ. Braunschweig, Germany, December 2004
19. Neumann, T., Koch, A.: A Generic Library for Adaptive Computing Environments. Intl. Conf. on Field-Programmable Logic and Applications (FPL), Belfast, Northern Ireland, UK, 2001
20. Lange, H., Koch, A.: An Execution Model for Hardware/Software Compilation and its System-Level Realization. Intl. Conf. on Field-Programmable Logic and Applications (FPL), Amsterdam, Netherlands, August 2007
21. Lange, H., Koch, A.: Memory Access Schemes for Configurable Processors. Intl. Conf. on Field-Programmable Logic and Applications (FPL), Villach, Austria, August 2000