PRECORE – A TOKEN-BASED SPECULATION ARCHITECTURE FOR HIGH-LEVEL LANGUAGE TO HARDWARE COMPILATION

Benjamin Thielmann

Integrated Circuit Design (E.I.S.)
Technische Universität Braunschweig
email: thielmann@eis.cs.tu-bs.de

Jens Huthmann, Andreas Koch

Embedded Systems and Applications Group
Technische Universität Darmstadt
email: {huthmann|koch}@esa.cs.tu-darmstadt.de

ABSTRACT

We propose a universal method to automatically generate both datapaths and the appropriate application-specific speculation-support logic from high-level C-language descriptions. Our approach aims to be lightweight by extending efficient statically-scheduled microarchitectures with a limited dynamic token model to predict, commit, and replay speculation events. As a first source of speculativeness, we evaluate the use of data-value speculation to speed-up memory reads when targeting a reconfigurable adaptive computer.

1. INTRODUCTION

Adaptive computing systems combine a software-programmable processor (SPP) with a reconfigurable compute unit (RCU). This combination of processing elements (PE) can efficiently improve the performance of many algorithms, with each processing element executing the parts of application it is best suited for.

However, even reconfigurable systems are affected by some of the basic problems in computer architecture, one being the memory bottleneck: Typically, 20% of the instructions are memory accesses, but they require up to 100x the execution time of the register-based operations [8].

The memory performance of the RCU can be improved by, e.g., providing it with a dedicated high-bandwidth path directly to the memory controller [11], by using a configurable multi-port caching/streaming memory system [10], by control speculation (higher priority to accesses whose data is more likely to be actually used) [12], or by localizing data to the fast on-chip memories of an FPGA [3]. However, whenever DRAM-based main memory is accessed, the RCU must be able to deal with variable latencies. This can be handled by a number of techniques: The classic approach (already used in VLIW compilation) statically schedules execution for the expected latency (e.g., a cache hit takes just one cycle), and stalls the entire PE otherwise until the read data is available. An alternative approach uses dynamic scheduling to stall just those parts of PE actually depending on a memory read, and allows the independent parts to continue [6]. While very flexible, such a dynamically scheduled microarchitecture requires more chip area due to the complexity of the scheduling and data/control synchronization logic.

As an alternative, we propose the use of *data speculation* on the RCU to hide the memory latency by making it appear that a read operation *always* returns data after a single cycle. With this approach, we could use an efficient purely statically scheduled microarchitecture on the RCU. However, since the data returned by the read might be incorrect, we now need to provide the RCU not only with the capability to *predict* a new data value to return, but also mechanisms to *commit* a computation once the predicted value has been proven to be correct, or to *replay* (re-execute) a computation with a new value once the initial prediction turned out to be false.

In this work, we introduce the general-purpose framework PreCoRe to address these issues for reconfigurable computing. PreCoRe supports both control and data speculation for custom-compiled reconfigurable datapaths.The basic approach is independent of the source of speculativeness in the system (data and address value speculation, memory dependence speculation, etc.). In contrast to the general-purpose speculation hardware used in dynamically scheduled super-scalar processors [8], PreCoRe employs application-specific speculation logic, generated by our high-level language to hardware compiler NYMBLE for each individual, also application-specific datapath. We aim to avoid slowing down the datapath compared to a non-speculative version (by not requiring additional clock cycles due to speculation overhead), even if all speculations fail continuously.

The PreCoRe mechanisms are lightweight extensions to a statically scheduled datapath, they do not require the full flexibility (and corresponding overhead) of a datapath dynamically scheduled at the level of individual operators (such as COCOMA [2]). Since speculation in general is only beneficial for long-latency operations, we have concentrated our efforts on (possibly cached) variable-latency

main-memory read operations.

We discuss the PreCoRe system here in a scenario where we have added data-value speculation to the read ports of the MARC II [12] reconfigurable multi-port cache-coherent memory system. For space reasons, this work concentrates on the commit/replay functionality of the framework. The actual value speculator (based on a multi-level finite context scheme), the NYMBLE compile-flow, and low-level hardware implementation details cannot be explained in greater detail.

2. RELATED WORK

Now that RCUs on modern FPGAs can directly interface to current high-speed memories such as DDR2/3-SDRAM, they are also affected by the processor/memory performance gap that has been plaguing SPPs for years [7]. But since RCU performance is heavily dependent on exploiting parallelism (usually fine-grained) with dozens or even hundreds of parallel operators, RCUs suffer a more severe performance degradation than SPPs (which generally have only 4...9 parallel execution units on a single core) when accessing external memories using the limited FPGA I/O bandwidth (compared to the high-bandwidth access to on-chip memories).

Data value speculation is a long-proposed technique to reduce the impact of the memory gap [14]. By using the speculated values to continue computation, long memory latencies can be hidden. However, as with all speculation/prediction schemes, actual gains can often only be achieved if the prediction is correct most of the time (classical branch prediction), or if the penalty to recover from misspeculations is very low (the PreCoRe approach).

Much research has been performed on data speculation methods and their accuracy. History based predictors, with the trivial case of a last-value predictor, predict the next value depending on the previously encountered values. Stride predictors can accurately predict sequences with a constant offset between elements. Context-based value predictors predict the value sequences for each load instruction individually [17]. The different schemes may also be combined. The limits of data value speculation have previously been discussed by Gonzáles et al. [5, 4].

However, the techniques have only seen very limited use in practice. As one of the few examples, Mock et al. modified a C compiler to force data speculation where possible on a Intel Itanium 2 CPU architecture [16]. The Itanium 2 roll-back mechanism is based on a dedicated hardware structure, the Advanced Load Address Table (ALAT), and usually needs to be explicitly controlled by the programmer [15]. The experimental results show that load value speculation often improves performance by as much as 10%. However, under adverse conditions with frequent misspeculations, performance losses of up to 5% have also been observed.

To our knowledge, PreCoRe is the first general-purpose framework for generating application-specific speculation-assist units from a high-level language targeting ACSs.

3. SPECULATION FRAMEWORK

PreCoRe uses two key mechanisms for its operation. The first is a token model for tracking the effects of speculative execution as well as means to commit and revert its effects. As a second component, specialized queues buffer both tokens and their associated data values, allowing the replay of failed speculation and the deletion of misspeculated data.

3.1. Token Mechanism

We will start with a simple statically scheduled datapath and extend it with PreCoRe token processing logic. Figure 1.a shows nine operators, organized into four Stages. In a pure statically scheduled datapath, a Stage would correspond to the clock cycle an operation executes in, e.g., the READ operation in Stage 1 would start at time t_0. However, assuming that this READ has a variable latency, e.g., is accessing a DRAM-based main memory via a cache using a system such as MARC II [12], pure static scheduling is no longer possible. Instead, operations are scheduled for their expected latency (one cycle, for the case of a cache hit). If that assumption does not hold (cache miss), the *entire* datapath is halted until the data is actually available. In the Figure, this is indicated by annotating the Stages by the time they actually start execution. Figure 1.a thus shows that while the READ is started at t_0, a cache-miss has delayed its result so that Stage 2 can only be started at a later time t_n ($n > 1$). Thus, the succeeding Stages 3 to 4 also start execution at later times t_{n+1} and t_{n+2}.

Stalling the entire datapath could be avoided by performing operator-level dynamic scheduling, which would allow, e.g., the right ADD in Stage 2 to proceed while only the READ stalled [1]. However, another approach is to *guarantee* that the READ returns data after a single cycle. This can be achieved by supporting load value speculation [18]: On a cache-miss, a speculated data value will be returned as read result. If additional speculation cannot proceed (e.g., due to exhausted queue capacity), the conventional method of halting the datapath until the access is resolved is used as fall-back.

The effect is shown in Figure 1.b. Here, the READ in Stage 1 executes at t_0 and immediately provides a speculated data value to the next Stage, which can then execute at t_1, and now also computes a speculative result for its successors. Assuming pipelining, at clock cycle t_3 the first speculative value (originating from the READ in Stage 1) reaches the WRITE in Stage 4. However, a memory write

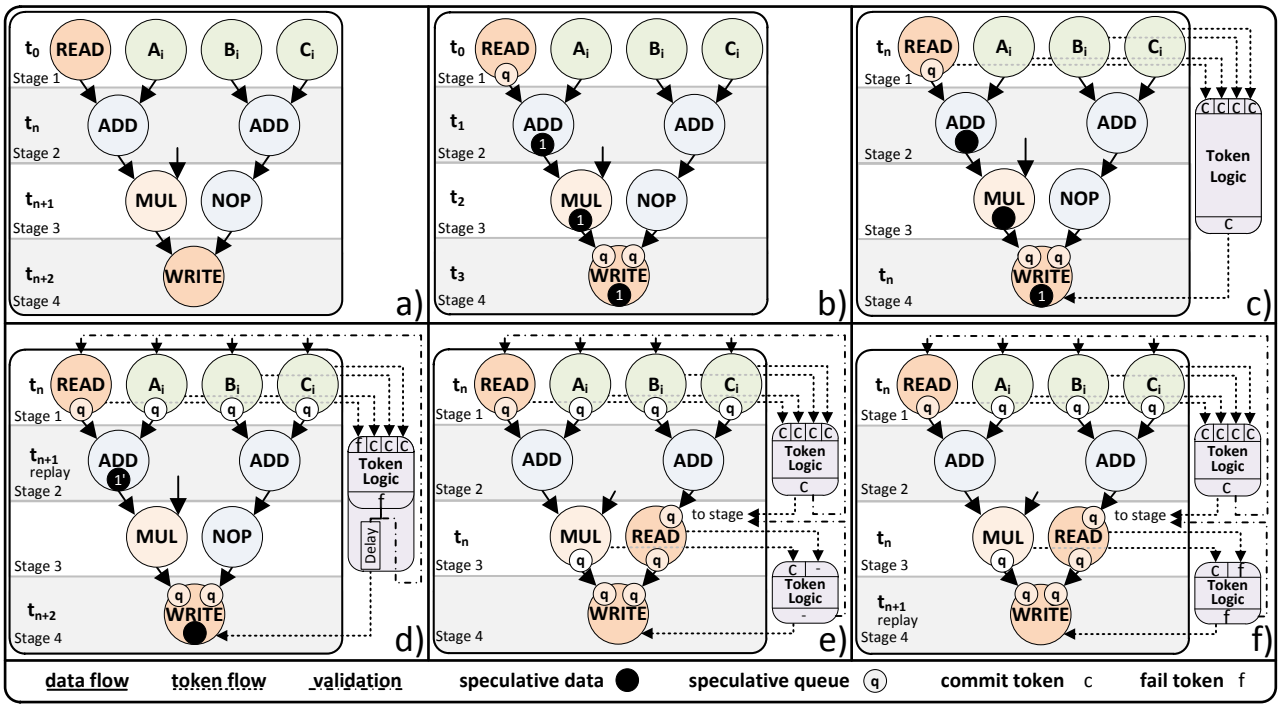

Fig. 1. Datapath and speculation token processing

is a destructive operation. Allowing it to proceed unchecked could result in the (possibly incorrectly) speculated value overwriting crucial data. A write operation thus forms the *speculation boundary* of the current approach (but see Section 5). For a WRITE, the system has to ensure that the data to be written has been correctly speculated in its originating READ node (which is the sole source of speculated data in the current system). For the READ, this is done by comparing the speculatively output result, which is retained for this purpose in an output queue in the READ node, with the actual data received later from the memory system. Until the result of that comparison, which determines the correctness of the speculation, is available, the data to be written (which depended on the speculative value) is held in an input queue in the WRITE node.

Figure 1.c extends the now speculative, but still statically scheduled datapath to process values correctly speculated for the READ. Since the speculated and actual data values have been determined to match at time t_n, the READ node indicates this by sending out a Commit-Token (shown as C in the Figure). In contrast to operator-level speculation, however, this is not directly forwarded to the WRITE node. Instead, the speculation status is computed per Stage: The speculation status signals of *all* operators in a Stage (even the non-speculative ones, since they might have operated on speculated values) are combined: Only if *all* operators in a Stage indicate the presence of correctly speculated data, is a corresponding C-Token forwarded to the operator at the speculation boundary (the WRITE). Here, it confirms the correctness of the last datum with undetermined speculation status (in the example, the speculated value originating in the READ node). Note that speculated values and their corresponding tokens are associated only by remaining strictly in-order, no additional transaction IDs or similar measures are required. However, tokens can temporarily overtake their corresponding datum up to the next synchronization point (see Section 3.2) , since the speculation status of an entire Stage is only affected if a speculative operator (in our approach, a READ node) is present there. Otherwise, the speculation status will just depend on the Stage inputs (non-speculative, if no speculative values were input; speculative, if even a single input to the Stage was speculative). Since the Stages 2 and 3 do not contain READs, the token can be directly forwarded at time t_n to the WRITE in Stage 4, where it will wait in a token queue for the correctly speculated value to arrive and allow the WRITE to proceed. Note that during this process, pipelining might have lead to the generation of more speculative values in the READ, which continue to flow into the succeeding Stages.

The case of handling a failed speculation is shown in Figure 1.d. Here, two actions have to take place: All data values depending on the misspeculated value have to be deleted (which also prevents the WRITE from executing), and the affected computations have to be restarted (replayed), this time with the correct non-speculative result of the READ. To this end, the token logic first determines that the misspeculated READ in t_n leads to the the entire Stage 1 failing in its speculation. In contrast to the C-Token, which had

an immediate effect, the Fail token (F-Token) is delayed by the number of stages between the source of the speculated value and the WRITE marking the speculation boundary (in this case, two Stages, leading to a delay of two clock cycles). To be more precise, the token is delayed by two clock cycles when the datapath is actually computing. If it were stalled (e.g., all speculative values have reached speculation boundaries, but could not be confirmed yet by memory accesses because the memory system was busy), these stall cycles would not count towards the required F-Token delay cycles. This ensures that all intermediate values computed using the misspeculated value in Stages 2 and 3 have now ended up in the input queues of the WRITE operation in Stage 4, and will be held there since no corresponding C- or F-Token for them has been received earlier. The appropriately delayed F-Token from the token logic then arrives at time t_{n+2} to resolve this situation, deleting the two sets of incorrectly computed intermediate results from the input queues and preventing the WRITE from executing. The replay starts at time t_{n+1} after the misspeculation has been discovered. READ outputs the correct data, all other nodes in Stage 1 output their last set of results (which may also be speculative themselves!) and perform the computations in Stages 2 and 3 again. These last results of operators are held specifically for replay purposes in per-operator output queues. Values are removed from theses queues only if *all* operators on a Stage have indicated to the token logic that their values are now correct by validating them using C-Tokens. In that case, one set of stored values can be safely discarded for the entire Stage, since a replay involving them will no longer be necessary.

Figure 1.e shows the case when multiple speculative data sources (READ operations at Stages 1 and 3) are present in the datapath. In this case, each READ begins its own speculation region, which ends either at a WRITE, or a Stage containing another READ. The system now has to handle the case that even though data has been committed in an earlier Stage (here, by the READ in Stage 1), the WRITE still cannot proceed since a later Stage is still speculative (READ in Stage 3). To this end, each of these regions has its own token logic, customized for the number of Stages in the region. The upper token logic would delay its F-Tokens by one cycle (that region begins at Stage 1 and has Stage 2 as an intermediate Stage before the end), the lower one would forward F-Tokens directly (its corresponding region, beginning at Stage 3, has no intermediate Stages, it ends at Stage 4). READ nodes beginning a new speculative region need to be provided with an input queue (for the address data), since the prior region might already have committed its data (and removed it from its queues), but the READ has not executed yet (e.g., due to lack of access to a shared memory bus). This is the scenario actually shown in Figure 1.e: MUL has already confirmed its output as being correct (due to Stage 1 confirming itself as correct), but the Stage 3 READ has not yet resolved its speculation status, which is thus pending for the entire Stage 3. In turn, the WRITE is not allowed to proceed. Furthermore, as usual with a Stage containing a READ node, *all* nodes in the Stage are provided with output queues to allow a replay should one of the Stage's read operations fail in its data speculation. Had the upper region failed due to misspeculated data in the Stage 1 READ, the F-Token from the upper token logic would delete the misspeculated values from the Stage 3 output queues, since they will be recalculated when the upper region is replayed. These output queues also serve to resynchronize values and their associated tokens, as tokens are not allowed to overtake their corresponding values across an output queue.

The scenario is advanced in Figure 1.f: Here, we assume that the Stage 3 READ has misspeculated at time t_n, leading to a replay of the succeeding Stages (in the example just the WRITE operation in Stage 4, which has been halted so far due to the unclear READ speculation status) starting at t_{n+1}. Since the intermediate results computed from values originating in Stages 1 and 2 have already been confirmed as being correct, the replay can be limited to just forwarding the now-correct values from the Stage 3 output queues to Stage 4. It is not necessary to re-execute the complete computation beginning with Stage 1.

3.2. Queue Management for Speculation

As described in the prior Section, operator queues play a significant role in the speculation mechanism. They are responsible for providing succeeding Stages with data (speculated or accurate), holding data until the need for a replay has been ruled out, and to discard misspeculated values. Furthermore, they synchronize the asynchronously flowing value and token sequences, as tokens cannot overtake values through an output queue. The queue's internal operation is thus somewhat more complicated than that of simple conventional queues.

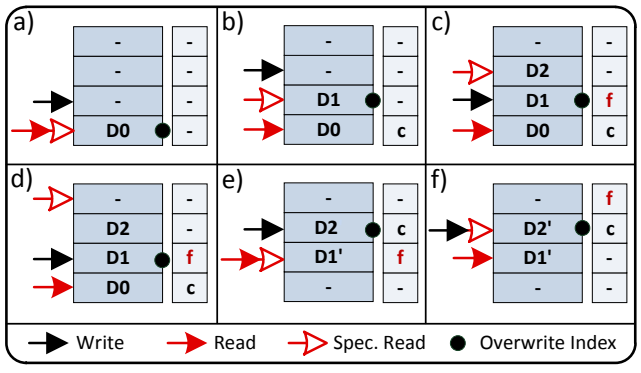

Fig. 2. Speculative output queue behavior

Figure 2.a shows the general organization of a specula-

tive output queue. Fundamentally, it consists of two circular buffers (one for values, one for tokens) organized as two queues: A Write pointer indicates the next available position to enter data, a Read pointer marks the first unread element. For token management (the right queue), this is the entire required functionality, the Figure thus shows the actual pointers only for the more complicated value queue (depicted at the left side in each subfigure). In the value queue, the data between the Read and Write pointers has either not been passed on to a succeeding operator, or its speculation status is as yet undetermined. To pass on speculative values to succeeding operators, the SpecRead pointer is used. All values below that pointer have been passed to succeeding operators, but they might be misspeculated values. The region starting with and extending above the OverwriteIndex are those positions holding known misspeculated values, they will need to be overwritten with new values (that may still be speculative).

We will illustrate the operation of the speculative output queue (and the interaction between the value and token queues) using an example execution sequence.

Figure 2.a shows the situation after a value D0 has arrived at the value queue. Write has advanced to the next available space, while all other pointers were not affected.

For Figure 2.b, three events have occurred concurrently: First, a second value D1 has arrived and is inserted into the value queue, advancing Write upwards. Second, D0 has been speculatively forwarded to a successor node, bumping up SpecRead. At the same time, a C-Token has arrived for the first value (here D0), confirming it as correct. In order to prevent it from being overwritten by a possible future replay of prior Stages, the OverwriteIndex is moved upward, protecting D0. Since the Stage holding this output queue has not been validated yet (other operators may hold uncommitted data), D0 as well as its corresponding C-Token are kept in the queue for a future replay of this Stage.

This is still the case in Figure 2.c. D0 and its C-Token are as yet unvalidated (e.g., another read operation at the same Stage has an uncertain speculation status). At the same time, D1 has been speculatively passed to a successor node, moving the ReadSpec pointer upwards, and a new value D2 has arrived and is inserted into the value queue. However, an F-Token incoming in the same cycle signals that the first uncertain value (being D1) was misspeculated. Thus, D1 and other values arriving since (here, D2) are considered incorrect and will be recalculated by a replay of the prior stages. To this end, Write is reset to the OverwriteIndex (which marks the boundary to the actually confirmed values).

In Figure 2.d, that replay has not occurred yet (e.g., due to a prior read operation not getting access to the shared memory bus), thus, no recalculated values arrive. However, the value queue continues to supply speculative (undefined) values to successor nodes, bumping up ReadSpec, since the F-Token indicating their incorrectness has not become visible yet at the head (bottom) of the token queue.

The situation is resolved in Figure 2.e, when the external token logic has validated this entire Stage. This leads to the removal of D0 (by moving Read upwards) and its corresponding C-Token (they will no longer be required for a replay starting at this Stage). The F-Token, now visible at the bottom of the token queue, indicates that starting from here the values speculatively forwarded to successors have all been incorrect. In order to begin forwarding the next set of re-calculated values, ReadSpec is reset to Read, and the F-Token, having performed its function, is removed. At the same time, the first new value D1', re-calculated by the replay in prior Stages, has arrived, and is immediately confirmed by a corresponding C-Token arriving in parallel. The value will stay in the queue until the entire Stage has been validated. Note that, if D1' had not arrived this cycle, the now-visible F-Token would have *prevented* the output of undefined values to the successors.

The final development (Figure 2.f), shows the situation a few cycles later: D1' is still unvalidated, but has been speculatively forwarded to successor nodes (ReadSpec has moved above it). However, the replay of earlier Stages has been misspeculating again. While it has re-calculated at new value D2', that value has already been marked with an F-Token. Thus, Write has been reset to OverwriteIndex, which will lead to the overwriting of the misspeculated value D2' with a result of the next replay of earlier Stages. No undefined values are output, since the special case of Write being equal to ReadSpec is recognized as an empty queue (regardless of the hidden F-Token).

As can be seen here, values and tokens are synchronized strictly by their sequential order, not by being in the same position of the queues. Furthermore, the token queue must have *twice* the length of the value queue to handle the corner case of an incoming sequence of alternating C- and F-Tokens.

Input queues behave similarly to output queues, with one exception: They do not require validation of committed data. Instead, values are removed from the queues as soon as their C-Tokens arrive.

4. EXPERIMENTAL RESULTS

Table 1 shows the results of executing a number of benchmark programs using our PreCoRe-enhanced NYMBLE compiler. The RTL Verilog created by NYMBLE was then synthesized for a Xilinx Virtex-5 FX device using Synopsys Synplify Premier DP 9.6.2 and Xilinx ISE 11.1. Our target ACS is based on the Xilinx ML507 development board, but extended with the MARC II high-performance memory interface [12]. As a baseline for our comparison (both for performance and area), we configured MARC II

Kernel	FPGA Area				Max. Clock Freq.		Runtime				Comparison	
	#LUTs		#Registers		(MHz)		#Cycles		μs at max. freq.		Slices	Speed
	n. spec	spec	n. spec	spec	n. spec	spec	n. spec	spec	n. spec	spec	ovrhd.	-up
array_add	10141	14717	1246	2948	106.90	96.30	6194	3923	57.94	40.74	1.45x	1.42x
bintree_search	11129	19782	1570	5795	105.70	100.10	3497	3359	33.08	33.56	1.78x	0.99x
gf_multiply	11918	22790	1702	6459	102.80	101.30	2510	2482	24.50	24.42	1.91x	1.00x
median_filter_row	12895	28333	2458	9909	106.40	102.20	296409	114650	2785.80	1121.82	2.20x	2.48x
median_filter_col	12998	28391	2529	10325	106.00	100.30	1054736	666554	9950.34	6665.24	2.19x	1.50x
pointer_chase	11979	20637	1478	10208	106.40	98.90	4087	3650	38.41	36.91	1.72x	1.04x
simple_read	10241	14703	1484	3639	105.80	103.20	19615	14150	185.40	135.41	1.45x	1.37x
versatility_quant.	12351	39716	5390	18495	105.70	97.40	96771	50746	915.53	521.01	3.22x	1.76x
versatility_fcdf22	12055	32284	1889	9863	105.20	93.80	43633	20573	414.76	219.33	2.68x	1.89x

Table 1. Hardware area, maximum frequency, and run-time without/with data speculation (n.spec/spec)

to provide cached accesses to the FPGA-external DDR2-SDRAM, with the memory ports being organized as a single coherency cluster. In the non-speculative case (using only static scheduling), we halt the entire datapath if a memory access cannot be resolved within a single cycle.

array_add increments each element of an array, without loop-carried dependencies. bintree_search searches a binary tree. gf_multiply is part of the Pegwit elliptic curve cryptography application and MediaBench [13]. median_filter_row and _col realize a luminance median filter. Blocks of 9 pixels are read row/column-wise and the median of luminance is written to the center pixel. The column-based processing shows the effectiveness of data speculation when cache efficiency decreases due to unsuitable access patterns. pointer_chase processes a randomly linked list, writing to every second element. simple_read sums all values of an array, and thus has a loop-carried dependency. versatility_quantization and versatility_fcdf22 are the quantization and Wavelet steps of an image compression benchmark [9].

As can be seen in the Table, enabling PreCoRe during hardware compilation carries an area overhead of 1.45x...3.22x (counting slices). This is due mostly to the current NYMBLE hardware back-end not exploiting the sharing of queues across multiple operators in a stage, and the pipeline balancing registers automatically inserted by the compiler not being recognized as mappable to FPGA shift-register primitives by the logic synthesis tool. Both of these issues can be resolved by adding the appropriate low-level optimization passes to NYMBLE. For some kernels, the added logic also leads to a drop in clock-rate of up to 11% over the non-speculative versions. However, since the system clock frequency of the ML507 board is limited to 100 MHz by the other SoC components, the worst clock slow-down observed amounts to just 6.2% in practice. Since most of the critical path lies inside of the MARC II memory system, the achievable maximum clock frequency is almost independent of whether a speculative or non-speculative execution model is chosen. Consequently, adding the single-cycle load speculation leads only to the observed limited drops in frequency.

Despite the current area and clock inefficiencies, enabling PreCoRe can achieve speed-ups for our benchmark applications of up to 2.48x. Compared to its non-speculative version at maximum theoretical clock frequency, only bintree_search would be slowed down (by less than 1%). When considering the actual 100 MHz system clock, the wall-clock improvements go up to 2.59x, and no slow-downs occur at all. These results show significant improvements over prior work (cf. Mock, see Section 2).

5. CONCLUSION AND FUTURE WORK

We have introduced an effective new general-purpose scheme for adding hardware-supported speculation to mostly statically scheduled datapaths. Our approach automatically generates the required structures when compiling from C for adaptive reconfigurable computers.

Future work will tackle both implementation weaknesses (e.g., reduce area overhead by merging of queues), as well as add new speculation sources, such as memory dependence speculation (which can dynamically reorder accesses). Furthermore, additional engineering effort will be expended on raising the maximum clock frequency to speed-up the entire SoC, specifically by deeper pipelining of the critical path(s) within the MARC II memory system. The increased latency should only have a limited impact on PreCoRe, since successful speculation still gives the appearance of single-cycle read accesses to the datapath.

Acknowledgements: This work was supported by the German national research foundation DFG and by Xilinx Inc. We would also like to thank the reviewers, who provided very valuable comments to guide our next research steps.

References

[1] H. Gädke and A. Koch. Accelerating Speculative Execution in High-Level Synthesis with Cancel Tokens. In *ARC*, vol. 4943/2008 of *LNCS*, pp. 185–195. Springer, 2008.

[2] H. Gädke-Lütjens. *Dynamic Scheduling in High-Level Compilation for Adaptive Computers*. Ph.D. thesis, Technical University Braunschweig, 2011.

[3] H. Gadke-Lutjens, B. Thielmann et al. A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Com-

pilation. *Intl. Conf. on Field Programmable Logic and Applications*, 0:475–482, 2010.

[4] J. González and A. González. The potential of data value speculation to boost ILP. In *Proc. Intl. Conf. on Supercomputing*, ICS '98, pp. 21–28. ACM, New York, NY, USA, 1998.

[5] J. González and A. González. Limits of Instruction Level Parallelism with Data Value Speculation. In *Intl. Conf. on Vector and Parallel Processing*, VECPAR '98, pp. 452–465. Springer-Verlag, London, UK, 1999.

[6] H. Gädke, F. Stock et al. Memory Access Parallelisation in High-Level Language Compilation for Reconfigurable Adaptive Computers. In *FPL*, pp. 403–408. 2008.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[8] D. Kaeli and P.-C. Yew. *Speculative Execution In High Performance Computer Architectures*. CRC Press, Inc., 2005.

[9] S. Kumar, L. Pires et al. A benchmark suite for evaluating configurable computing systems—status, reflections, and future directions. In *FPGA*, pp. 126–134. ACM, New York, NY, USA, 2000.

[10] H. Lange and A. Koch. Memory Access Schemes for Configurable Processors. In *FPL*, pp. 615–625. 2000.

[11] H. Lange and A. Koch. Architectures and Execution Models for Hardware/Software Compilation and their System-Level Realization. *IEEE Trans. on Computers*, 99(PrePrints), 2009.

[12] H. Lange, T. Wink et al. MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers. In *DATE*. 2011.

[13] C. Lee, M. Potkonjak et al. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems.

[14] M. H. Lipasti, C. B. Wilkerson et al. Value Locality and Load Value Prediction. pp. 138–147. 1996.

[15] C. McNairy and D. Soltis. Itanium 2 Processor Microarchitecture. *IEEE Micro*, 23:44–55, 2003.

[16] M. Mock, R. Villamarin et al. An Empirical Study of Data Speculation Use on the Intel Itanium 2 Processor. In *Proc. Workshop on Interaction between Compilers and Computer Architectures*, pp. 22–33. IEEE Computer Society, Washington, DC, USA, 2005.

[17] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proc. Intl. Symp. on Microarchitecture*, MICRO 30, pp. 248–258. IEEE Computer Society, Washington, DC, USA, 1997.

[18] B. Thielmann, J. Huthmann et al. Evaluation of Speculative Execution Techniques for High-Level Language to Hardware Compilation. In *ReCoSoC*. 2011.