RAP: MORE EFFICIENT MEMORY ACCESS IN HIGHLY SPECULATIVE EXECUTION ON RECONFIGURABLE ADAPTIVE COMPUTERS

Benjamin Thielmann

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

Thorsten Wink, Jens Huthmann, Andreas Koch

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

ABSTRACT

Increasing the degree of speculative execution in application-specific microarchitectures, which can be generated for reconfigurable computers from high-level code using techniques such as PreCoRe, also leads to an increased pressure on the memory system. The RAP approach introduced here describes and evaluates application-specific microarchitectural techniques to reduce the impact of aggressively speculated memory accesses. It covers a light-weight resolution mechanism for dynamic RAW memory dependencies, avoiding execution replays due to misspeculated reads, and a prioritization scheme for arbitrating the use of shared resources based on the degree of speculativeness of the individual access.

I. INTRODUCTION

Adaptive Computing Systems (ACS) combine a Software-Programmable Processor (SPP) with a Reconfigurable Computing Unit (RCU). This heterogeneous combination of Processing Elements (PE) can efficiently improve the performance of many algorithms, with each PE executing the parts of an application it is best suited for. Commonly, the SPP executes general-purpose control and I/O operations not on the critical path, while the performance-critical "hot paths" (often called *kernels*) are realized in dedicated application-specific microarchitectures on the RCU.

Despites their advantages, ACS architectures are still a niche technology, as their programming requires experience in digital hardware design, computer architecture, and hardware description languages (Verilog/VHDL). To make the potential of ACSs available to a wider user base, considerable research effort (e.g., [2], [14], [10]) has been expended on automatic compilers for such heterogeneous computers. A new wave of commercial programming tools [1], [16], [17] allows the description of microarchitectures for an RCU using subsets of high-level programming languages such as C or SystemC to describe individual kernels.

All of these tools concentrate on extracting performance from single-threaded code by exploiting Instruction-Level Parallelism (ILP) when generating the hardware architecture for the RCU. However, the increased ILP in an RCU (not bounded by classical software instruction windows), leads to an even greater impact of the memory bottleneck: 20% of all instructions are memory accesses, but they require up to 100x the execution time of register-based operations [9].

Numerous techniques for improving the memory performance of RCUs having high internal parallelism have been proposed: On-chip local memories are available in sufficient quantity on modern reconfigurable devices to directly provide the required degree of parallel accesses. However, their small storage capacity necessitates careful allocation by the human designer or the compiler, often also requiring swapping/paging between local and main memory [4]. Other approaches attempt to increase the raw memory bandwidth available to the RCU [12], or support both parallelism as well as better bandwidth utilization by configurable multi-port caching/streaming memory systems [11].

Another aspect that needs to be considered is the variable latency of accessing cached or DRAM-based memory. Traditionally, this has been handled (e.g., in VLIW architectures) by stalling the entire execution until the requested data becomes available. Alternatively, dynamic scheduling can be used to stall just those parts of a computation that actually depend on the read data, and to allow independent parts to proceed. For RCUs, such an approach has been described in [3]. While more flexible, it carries a significantly higher cost in hardware area.

An orthogonal approach aims to guarantee fixed latency accesses. While this could be achieved, e.g., by only using SRAM, an architecture such as PreCoRe [19], [18] guarantees single-cycle read operations on DRAM-based main memory by the use of *value speculation*: A datum is always provided within a single clock cycle, either from a cache hit, or by predicting the next read value on a cache miss. As these predictions may turn out to be incorrect, the microarchitecture must be extended to re-execute the

affected parts of the computation with the correct operands and commit only those results computed from values that were either correctly speculated or actually retrieved from memory. Since intermediate computations can be affected by multiple reads (which are the sole sources of predicted values), the scope of the re-executed operations must be determined with some care: Ideally, only those computations actually affected (sometimes called "poisoned") by the misspeculated value need to be replayed. While this could be handled at the granularity of individual operators [10], PreCoRe aims for a more efficient hardware implementation by performing predicts, commits, and replays at the granularity of *stages*, which correspond to the clock cycles of an initially purely statically scheduled datapath.

The actual performance of PreCoRe is highly dependent on the data speculator's accuracy. Furthermore, the aggressive speculation imposes even higher requirements on the memory system to support both speculative and non-speculative prefetching [18]. On the other hand, the prefetching efficiency is adversely affected by the presence of memory dependencies. If they cannot be disproven statically (e.g., by alias / points-to-analysis), they need to be handled at run-time. This can be achieved either by executing the memory accesses strictly in program order, thus ensuring the adherence to the original dependencies, or by allowing out-of-order execution. In the latter case, a greater degree of parallelism can be achieved, but dependency violations have to be detected and corrected.

For efficiently handling these issues, this work introduces **RAP**, a collection of techniques to **R**esolve, **A**void, and **P**rioritize out-of-order memory accesses with dynamically occurring memory dependencies. RAP augments our prior work on PreCoRe, which concentrated on the microarchitectural enhancements required for the speculative commit/fail and replay mechanisms, the actual data value prediction techniques, and a study of the interaction of different speculation effects. Furthermore, RAP ensures that memory dependencies are enforced by re-executing parts of a computation which were dependent, but also recognizes independent accesses and allows them to proceed in parallel. Accesses to shared resources such as main memory are sequentialized by a priority scheme that takes their degree of speculativeness into account (accesses whose data is more likely to be used will get a higher priority). This technique, which was initially introduced as an extension of a multi-port cache in [13], is now integrated into the RAP suite.

II. RELATED WORK

While both SPPs and RCUs rely on pipeline parallelism for performance, only RCUs can aggressively exploit ILP to accelerate single-threaded programs even further. By following the paradigm of *spatially distributed computation*, an RCU is not limited to the $4 \ldots 8$ execution units of superscalar SPPs, but can implement as many parallel hardware operators as required by the algorithm. In many cases, this can lead to a significant performance advantage of the RCU over SPP despite the considerably lower clock frequency used on the reconfigurable device. Structural hazards would occur on an RCU only if the available reconfigurable capacity is exhausted. A more severe limit on the achievable ILP are memory data dependencies: According to [7], in practice only tens to at most hundreds (in very few applications) of instructions are free of dependencies.

Lipasti et al. proposed load value speculation to resolve read data dependencies speculatively [15], allowing computations to continue using the speculated values without waiting for the memory system to return the actual (possibly cached) memory contents. Since the speculated data could be used as address of another read or write, or as the data of a write, a single speculated read could start an entire cascade of dependent memory operations which now execute with value-speculated operands.

For control speculation, different mechanisms are used in SPPs and RCUs: On an SPP, branch-prediction could decide to speculatively execute a memory access on the branch most likely taken. But on the RCU, commonly all branches of a conditional are executed in parallel using dedicated hardware operators. On the other hand, the spatially distributed computation paradigm no longer applies once singleton resources (of which only a single instance can exist), such as the main memory, are considered. Such a resource has to be *shared* among the individual operators accessing it in parallel. On an SPP with branch prediction, this contention would not occur, as only a single branch alternative would execute. It is thus worthwhile to examine how to employ a mechanism similar to branch prediction when accessing a shared singleton resource from the RCU. Our approach builds on technology initially used in two-level adaptive branch predictors for SPPs as proposed by Yeh and Patt [20]. We will employ a similar technique to prioritize access to a shared resource if it occurs in the parallel branch most likely to be selected next [13].

Load value and control speculation cannot be considered completely independently: Value-speculated data may be evaluated in a control condition, while control-speculated parallel read operations may lead to a greater presence of speculated load values in the system (since the shared main memory cannot supply the actual values for all alternative branches simultaneously). Thus, the large body of prior research that considered data and control speculation methods and their accuracy separately is insufficient to fully gauge the effects on the memory system, on which only few works have focused at all [6], [5].

To our knowledge, PreCoRe [19], [18] is the first general-purpose framework *combining* control- and load value-speculation in application-specific microarchitectures

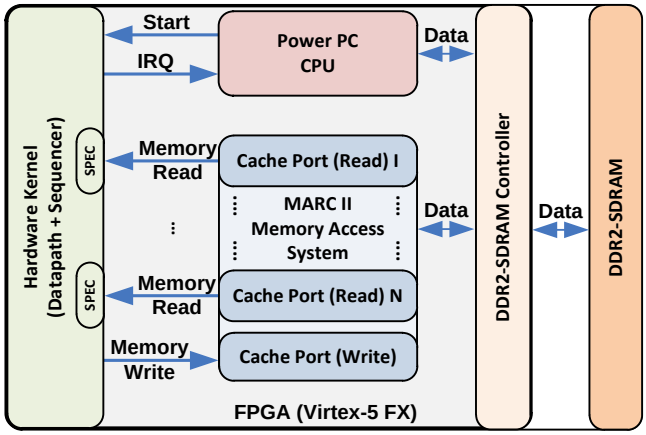

Fig. 1. ACS based on the Xilinx ML507 Virtex 5 FX FPGA Board

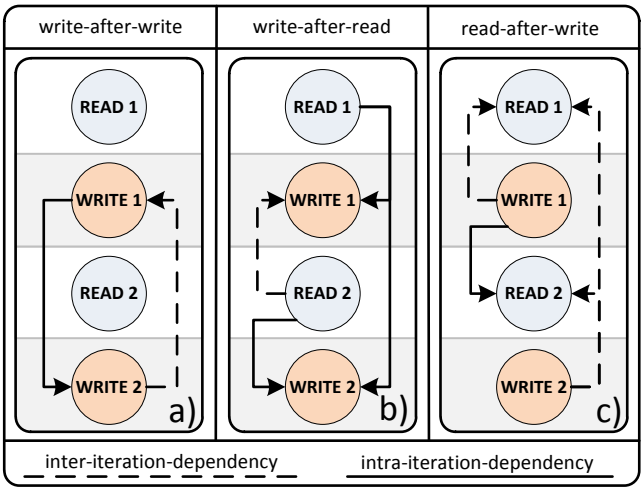

Fig. 2. Memory Dependencies

compiled from high-level languages. With the RAP extensions presented here, we aim to reduce the significantly increased memory pressure induced by the aggressive speculation.

III. TARGET ARCHITECTURE

The ACS we use for evaluating the techniques proposed here is based on a Xilinx ML507 development board. The superscalar PowerPC 440 core embedded in the Virtex 5 FX FPGA acts as SPP, the rest of the FPGA's reconfigurable fabric as RCU. The RCU has a dedicated FastLane+ high-bandwidth interface to the DDR2-SDRAM controller considerably more efficient than the stock PLB-based memory interface. In conjunction with other hardware extensions and a customized version of the full-scale Linux operating system, RCU-SPP signaling latency using the FastPath modifications has been improved by up to 23x even over a kernel extended with real-time patches. Furthermore, supported by AISLE extensions, the RCU can access the shared memory in the same virtual address space as software on the SPP. All of these enhancements are described in greater detail in [12].

A hardware kernel executing on the RCU usually does not access the raw FastLane+ memory interface directly. Instead, the MARC II [13] memory system is inserted between the kernel and FastLane+ (Figure 1) to provide multiple caching/streaming memory ports with configurable coherency and organization (e.g., stream buffer size, cache lines and line length, etc.).

This work will concentrate on efficiently using multiple cache ports in parallel. In MARC II, each port has a dedicated direct-mapped L1 cache. The ports have single cycle latency on a cache hit and support pipelined memory accesses for high throughput. Ports are organized in an arbitrary number of clusters. Within a cluster, cache-coherency between its ports is automatically maintained. A cluster may hold an arbitrary number of read ports, but only a single write port. Memory accesses proven to be independent by static analysis (accessing non-overlapping address ranges) can be assigned to different clusters.

Specifically, we will be examining how to use RAP to make the more difficult case (access independence not statically provable, all ports in a single cluster) more efficient. This requires dynamic resolution of memory-dependent accesses as well as the serialization of multiple kernel write accesses through the cluster's single write port.

IV. ENFORCING MEMORY DEPENDENCIES

The simplest way of enforcing the different kinds of memory dependencies Write-After-Write (WAW), Write-After-Read (WAR), and Read-After-Write (RAW), consists of just executing them sequentially in program order, even on the spatially distributed computing microarchitecture of an RCU. However, this leads to a significant drop in the ILP so crucial for RCU performance gains. It is thus worthwhile to examine other approaches. Specifically, we consider dependencies in context of a pipelined inner loop. As shown in Figure 2, dependencies exist both *forward* within a loop iteration and *backward* from subsequent loop iterations. Example: For the WAW dependency shown in Figure 2.a, WRITE1 has to execute *before* WRITE2 within a single iteration, but WRITE1 of iteration $n+1$ has to execute *after* WRITE2 in iteration n.

Given the MARC II memory system described above, WAW dependencies between write operations are always enforced, since the writes are time-multiplexed in program order onto the single write port of a coherency cluster.

The correct handling of WAR dependencies in PreCoRe [18] is a side-effect of committing only complete *stages* instead of individual operators: Even when speculatively ignoring their intra-iteration dependency in Figure 2.b, the

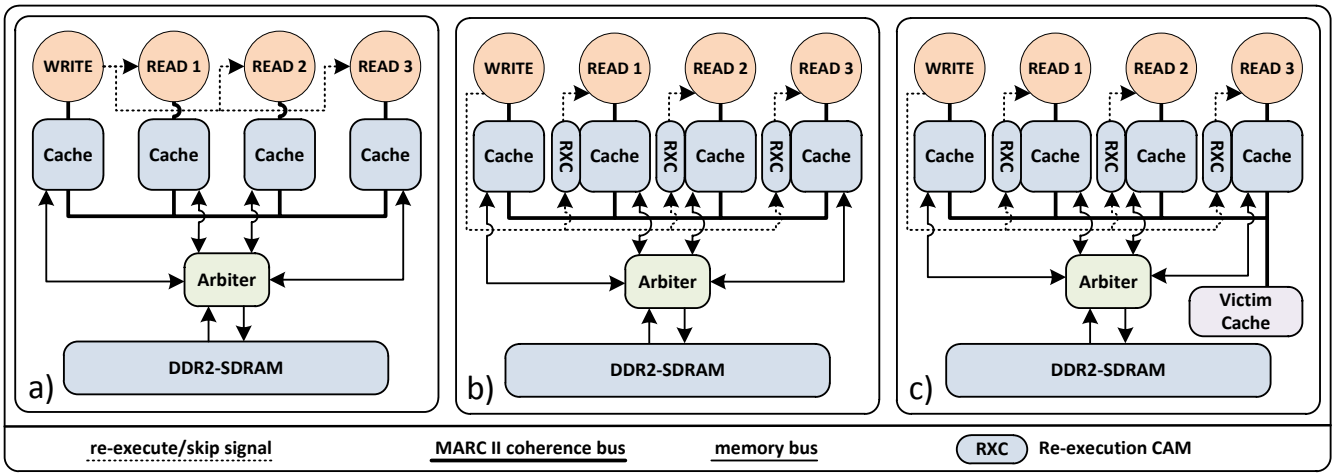

Fig. 3. Mechanisms for RAW dependency resolution in the MARC II cache system

stage holding WRITE2 will only execute once the prior stage holding READ2 has completely been committed.

PreCoRe alone was not able to handle RAW dependencies so far and supported only applications with non-overlapping read/write address ranges. To improve this, RAP will need to handle a case such as shown in Figure 2.c: The stage holding WRITE1 has been committed, but the write itself is still in progress. Since PreCoRe assumes a lack of dynamic RAW dependencies, it currently starts the stage holding READ2 (which might actually have begun even earlier to perform a speculative pre-fetch), overlapping completion of the write with the read operation. If both operations accessed the same address (=actually have a RAW dependency), the read will have fetched stale data, violating the dependency. RAP will need to extend the PreCoRe re-execution mechanism to replay the read (and the parts of the computation that are data-dependent on it) with the value actually written. Note that the extent of the required re-execution can be limited by considering the dependency direction: In Figure 2.c, WRITE1 will never affect READ1 in the first iteration, but could in subsequent iterations.

V. DYNAMIC RESOLUTION OF RAW DEPENDENCIES

Two issues must be considered when enhancing PreCoRe for handling RAW dependencies. First, PreCoRe's existing commit/replay scheme can be used as the base for such functionality, but must be extended to keep reads which ignore RAW dependencies speculatively, until all prior writes they could possibly depend on (even across iteration boundaries) have committed. The resulting mechanism is responsible for dynamically **R**esolving dependencies in **R**AP.

Second, efficient replays depend on the correct data being available in one of the caches, thus being accessible using the intra-cluster coherency scheme without the need to go to off-chip memory. Since speculatively executing reads without regard for potential RAW dependencies can cause even more replays (if the dependencies actually did exist), it becomes crucial to keep the correct data in the caches. This function is the **A**voidance of memory accesses in RA**P**.

V-A. Resolution Mechanisms

Universal Replay: This approach (shown in Figure 3.a) is a straightforward, low area, but sub-optimal extension of the existing PreCoRe commit/replay mechanisms: *All* RAW-speculated reads re-execute as soon as all writes have completed, regardless of whether an address overlap occurred. The number of affected reads is only limited by the PreCoRe speculation depth, which is the number of potentially incorrectly speculated intermediate results that can be rolled back. In practice, it is the depth of speculation value queues between PreCoRe read and write operators.

Assuming that the data written by all write operations is still present in the write cache, the replays will be very quick: All reads will get cache hits using the intra-cluster coherency scheme. Furthermore, if control-speculated writes get canceled once the misspeculation is discovered, the reads also need to be informed (using a Skip signal) that no dependency can be triggered from these writes and they can be ignored. If all writes are skipped, the read replay can be completely avoided.

Selective Replay: This more refined technique avoids unnecessary replays by detecting read/write address overlaps and replaying only those RAW-speculated reads actually affected by writes. To this end, MARC II must be modified as shown in Figure 3.b): First, each cached read-port is extended with a (relatively small) Re-Execution CAM (RXC) that holds the last n requested read addresses, where n is the PreCoRe speculation depth. Second, the

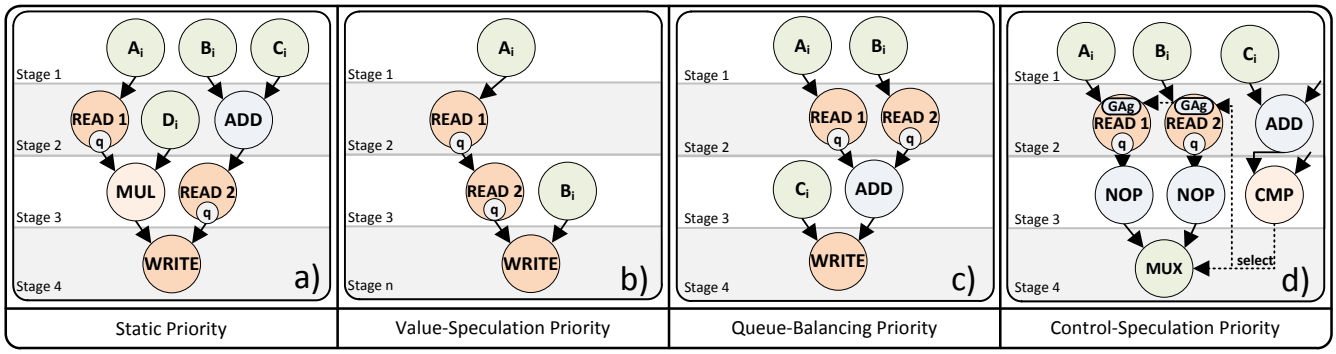

Fig. 4. Scenarios for priority-based shared resource arbitration

write port uses the existing coherency bus to broadcast write addresses to all read ports. If a RAW-speculated read was performed for an address overlapping a write address (as determined by a RXC lookup), a RAW violation is detected and that read needs to be replayed to deliver the newly written data. As before, the Skip signal is still required to ignore the potential dependencies of canceled writes.

V-B. Avoiding Memory Accesses during Replay

To allow fast replays, it is important to retain newly written data in the MARC II caches. Some obvious steps to achieve this would have high hardware or performance costs. Larger caches would require significant chip area, since the cache coherency scheme in MARC II relies on all caches having the same size, and thus an increase of the cache size would be multiplied by the number of ports. A fully associative cache would have fewer conflict misses and thus be less likely to evict important data. However, the CAMs associated with larger caches not only require significant hardware area, they also have long delays in reconfigurable logic. For these reasons, the fully associative caches originally used in MARC I [11] have been replaced by the smaller and faster direct-mapped caches of MARC II.

A better solution for our specific requirements is the addition of a small, but fully associative victim cache [8] that buffers blocks displaced from the per-port caches (Figure 3.c). By sizing it for the current speculation depth, we can ensure that newly written data is not lost even if conflict misses occur in the port caches. For typical speculation depths of 8 or 16, the required CAMs are both small and fast. By using a shared victim cache, we can avoid the additional overhead of keeping per-port victim caches coherent. Instead, the intra-cluster coherency bus can be used by the shared victim cache to communicate with the distributed caches.

VI. ACCESS PRIORITIZATION

With the distributed per-port caches of MARC II (see Figure 3), the spatial computing paradigm is extended to some parts of the main memory system. However, at some point, time-multiplexed access to shared singleton resources such as intra- and inter-cluster buses, the victim cache as well as the DDR2-SDRAM itself becomes necessary. By carefully **P**rioritizing different kinds of accesses using RA**P**, the negative performance impact of such time-multiplexing can be reduced.

VI-A. Static Priority

Even without RAP, MARC II supports priorities by statically assigning memory operations to the appropriate port. Accesses can thus be arbitrated to better fit the requirements of the datapath: The write port in a cluster always executes with the highest priority since it will only be fed with non-speculative data. Read operations scheduled early will be assigned to read ports with lower port numbers (having a higher priority), so that their data is already available when later stages execute. In Figure 4.a, READ1 thus executes with higher priority than READ2.

VI-B. Value-Speculation Priority

Figure 4.b shows a scenario where the address of READ2 is dependent on the result of READ1. In PreCoRe, READ1 will provide a value-speculated result after a single clock cycle, which READ2 will use as address for prefetching. However, in doing so, it will hog the shared MARC II resources performing a potentially useless access (if READ1 misspeculated), but possibly preventing the non-address speculated READ1 of the *next* iteration, an access that will be always useful, from executing. Value-Speculation Priority dynamically lowers the priority of accesses operating on speculated address and/or data values, thus giving preferential treatment to accesses using known-correct operands.

VI-C. Queue-Balancing Priority

In some situations, the simple static per-port priority can lead to a loss of performance. Specifically, this occurs if the outputs of multiple reads in the same stage converge at a later operator. An example for this shown in Figure 4.c. Here, the static priority would always prefer the read

assigned to the lowest port number over another one in the same stage. Assuming READ1 had the lower port, it would continue executing until its output queue was full. Only then would READ2 be allowed to fetch a single datum. A better solution is to dynamically lower the priority of reads already having a higher fill-level of non-speculated values (=actually fetched from memory) in their output queues.

VI-D. Control-Speculation Priority

As described above, read operators may output a speculated value when started. Orthogonal to this is speculating whether to execute the read operator at all. Such control-speculation is performed on SPPs using techniques such as branch prediction. While this approach is not applicable in the spatially distributed computation domain of the RCU (all ready operators execute in parallel), it does have advantages when dealing with shared singleton resources such as main memory: On an SPP, branch prediction would execute only the most likely used read in a conditional, while the reads on all branches of the conditional would be executed in parallel on the RCU, thus competing heavily for the shared resources and potentially slowing down the overall execution (on multiple parallel cache misses).

To alleviate this, we track which branch of a parallel conditional actually performed useful computations in light of the evaluated control condition. The read operators in that branch will receive higher priorities, thus preventing reads in lesser-used branches from hogging shared resources. To this end, we use decision tracking mechanisms well established in branch prediction, specifically the GAg scheme [20], but add these to the individual read operators of the parallel conditional branches (see Figure 4.d). The trackers are connected to the controlling condition for each branch (see [13] for details) and can thus attempt to predict which branch will be useful next from the past branching history, prioritizing its read operators. In case of a misprediction, all mistakenly started read operations are quickly aborted to make the shared resources available for the actually required reads.

VI-E. Composite Dynamic Priority Calculation

The dynamic priority $\mathsf{P}_{\mathsf{dyn}}(r)$ for each read operator r is thus computed from the queue-balancing priority $\mathsf{P}_{\mathsf{q}}(r)$, the control-speculation priority $\mathsf{P}_{\mathsf{hist}}(r)$ based on its GAg predictor, and its speculative predicate $\mathsf{IsSpec}(r)$, which is true if r is dynamically speculative for *any* reason (input address speculated and not yet confirmed, control condition not yet evaluated, still outstanding writes for RAW dependency checks).

$$\mathsf{P}_{\mathsf{dyn}}(r) = (\mathsf{W}_{\mathsf{q}} \cdot \mathsf{P}_{\mathsf{q}}(r) + (1 - \mathsf{W}_{\mathsf{q}}) \cdot \mathsf{P}_{\mathsf{hist}}(r)) >> (\mathsf{W}_{\mathsf{spec}} \cdot \mathsf{IsSpec}(r))$$
$$\Leftrightarrow \mathsf{P}_{\mathsf{dyn}}(r) = \mathsf{P}_{\mathsf{qh}}(r) >> \mathsf{S}(r)$$

where the W_{x} are static weights that can be set on a per-application basis, potentially even automatically by sufficiently advanced analysis in the compiler. W_{q} is used to trade-off between queue balancing and control history prediction, while $\mathsf{W}_{\mathsf{spec}}$ determines the priority penalty for speculative accesses. For brevity, the part of the expression dealing with priorities is termed $\mathsf{P}_{\mathsf{qh}}(r)$, the part dealing with penalizing speculation $\mathsf{S}(r)$, and $>>$ indicates a right-shift.

Note that the cluster write port, operated with purely static priority, will always have precedence over any read ports. The impact of the weights, as well as examples for the actual number ranges of the different quantities, will be considered in the experimental evaluation of the scheme in Section VII-B.

VII. EXPERIMENTAL RESULTS

In this section we will evaluate the different components of RAP for different applications and parameter sets. All of the hardware features described have been mapped to and measured on the actual ACE M5 ACS platform (see Section III).

VII-A. Dynamic Resolution of RAW Dependencies

In order to evaluate the different resolution mechanisms, we implemented a synthetic benchmark application (using our C-to-hardware compiler Nymble) which performs a mix of speculated accesses (control, value, RAW-dependency). Furthermore, we control the number of conflict misses in the direct-mapped MARC II caches by reducing the effective number of the cache port's 1024 lines to just 16, 8, and 4 lines. We will use this to observe the impact of victim caches on the dependency speculation schemes. Roughly a third of the read accesses will actually cause replays due to violated RAW dependencies.

Table I measures the speed-up of using the different techniques relative to a non-speculative execution without a victim cache (speed-up = 1.0). Alternatives examined vary in the number of effective caches lines (lower numbers induce more conflict misses), the Replay Mode used (with "always" indicating Universal Replay, but not able to ignore misspeculated writes using the Skip signal), and $\{0, 16, 32, 64\}$ entries in the shared victim cache. For each execution, we show the number of reads completed, the total number of reads issued (including the misspeculated ones that will get canceled before completing), and the runtime in clock cycles. We also give the relative areas and clock frequencies compared to the non-speculative baseline version.

The benchmark application significantly profits from speculative execution due to the increased ILP (up to 1.58x cycle-based speedup). The victim cache alone contributes to the speed-ups by 5% to 20%, since it compensates for conflict misses. However, the majority of the speed-up is due to the speculation mechanisms. Even the simple always scheme achieves speed-ups of $16 \ldots 28\%$ without a

shared victim cache. Further improvements by employing the smarter Universal and Selective replay methods yield gains of 39…46%, again without a victim cache. The impact of the better methods increases as replays become more expensive due to more conflict misses. The simple always scheme profits significantly from the victim cache (up to 18%), but even the better methods gain up to 9% of speed-up. The number of entries in the victim cache can be relatively small, with the theoretical upper bound being the number of read ports times the speculation depth. This would cover the worst case of all read ports continuously misspeculating and always evicting a line that will be required in a later replay. All of these lines would then be caught in the victim cache and allow fast replays without main memory accesses. However, even larger victim caches eventually lead to an unbalanced architecture and corresponding slow-downs (due to excessive speculation no longer supported by other parts of the memory system).

The current implementation of the system has focused on features and correctness, less on a highly optimized mapping. Thus, we observe a clock frequency slow-down of up to 8% over the non-speculative baseline kernels, mainly due to the more complex speculation logic having longer delay paths. While we still achieve net wall clock time speedups even with this slowdown, future work will involve refining the implementation for both smaller area and higher clock frequencies.

VII-B. Access Prioritization

We evaluate the access prioritization on three different benchmark applications, representing the different cases shown in Figure 4. Table II shows the results for a number of configurations of the dynamic priority scheme, expressed as % speed-up over an execution without dynamic priority.

The column Simple shows the impact of using a simpler dynamic prioritization formula

$$\mathsf{P}_{\mathsf{simple}}(r) = \mathsf{P}_{\mathsf{max}} >> \mathsf{S}(r)$$

that just penalizes all speculative accesses, regardless of their queue fill states and conditional execution history (represented by using a constant maximum priority $\mathsf{P}_{\mathsf{max}}$, see below). If even the speculative accesses are ignored by also setting $\mathsf{W}_{\mathsf{spec}} = 0$, that entry represents an execution without dynamic priority.

In the actual hardware, all arithmetic is performed in 6b unsigned integer arithmetic with normalized values ($63 \equiv 1.0$). Given that $\mathrm{F_q}(r)$ is the fill level of a read r output queue, which has 16 elements for speculation depth 16, the normalized queue-balancing priority is $\mathrm{P_q}[5{:}0](r) = (63 \cdot (16 - \mathrm{F_q}(r)))/16$. The normalized control-speculation priority is $\mathrm{P_{hist}}[5{:}0](r) \in \{000000, 010101, 101010, 111111\}$ for predicted branch execution probabilities of $\{0.0, 0.33, 0.66, 1.0\}$. Since the weights $\mathrm{W_q} = 1 - \mathrm{W_{hist}}$ are set statically at compile time,

Effect. Lines	Replay Mode	Vict.-Cache	#Proc. reads	#Issu. reads	Runtime [cycl.]	Speed-up	Area Ovhd.	Clk Speed
16	non.spec.	0	200	200	12598	1.00	1.00	1.00
16	non.spec.	16	200	200	12298	1.02	1.01	1.00
16	non.spec.	32	200	200	11747	1.07	1.01	1.00
16	non.spec.	64	200	200	11418	1.10	1.03	1.00
16	always	0	1955	2292	10828	1.16	1.80	0.95
16	always	16	2932	3442	9263	1.36	1.80	0.95
16	always	32	1865	3652	9207	1.37	1.81	0.95
16	always	64	1971	3727	9203	1.37	1.82	0.95
16	universal	0	885	857	9082	1.39	1.80	0.95
16	universal	16	748	1007	8851	1.42	1.80	0.95
16	universal	32	885	1114	8820	1.43	1.81	0.95
16	universal	64	972	1190	8882	1.42	1.82	0.95
16	selective	0	531	756	8839	1.43	1.83	0.92
16	selective	16	670	880	8692	1.45	1.83	0.92
16	selective	32	801	979	8667	1.45	1.84	0.92
16	selective	64	894	1031	8693	1.45	1.85	0.92
8	non.spec.	0	200	200	13890	1.00	1.00	1.00
8	non.spec.	16	200	200	13362	1.04	1.01	1.00
8	non.spec.	32	200	200	12293	1.13	1.01	1.00
8	non.spec.	64	200	200	11790	1.18	1.03	1.00
8	always	0	1712	2292	10828	1.28	1.80	0.95
8	always	16	2393	2938	9997	1.39	1.80	0.95
8	always	32	3139	3354	9339	1.49	1.81	0.95
8	always	64	3209	3412	9376	1.48	1.82	0.95
8	universal	0	832	1188	9713	1.43	1.80	0.95
8	universal	16	956	1312	9429	1.47	1.80	0.95
8	universal	32	1020	1372	9207	1.51	1.81	0.95
8	universal	64	1624	1956	9356	1.48	1.82	0.95
8	selective	0	328	589	9540	1.46	1.83	0.92
8	selective	16	313	767	9297	1.49	1.83	0.92
8	selective	32	836	1038	9132	1.52	1.84	0.92
8	selective	64	942	1137	9128	1.52	1.85	0.92
4	non.spec.	0	200	200	14272	1.00	1.00	1.00
4	non.spec.	16	200	200	14154	1.01	1.01	1.00
4	non.spec.	32	200	200	12548	1.14	1.01	1.00
4	non.spec.	64	200	200	11861	1.20	1.03	1.00
4	always	0	535	1140	11325	1.26	1.80	0.95
4	always	16	614	1207	11050	1.29	1.80	0.95
4	always	32	2822	2408	9559	1.49	1.81	0.95
4	always	64	2875	2534	9570	1.49	1.82	0.95
4	universal	0	371	714	10033	1.42	1.80	0.95
4	universal	16	412	747	9898	1.44	1.80	0.95
4	universal	32	803	1110	9197	1.55	1.81	0.95
4	universal	64	1003	1292	9418	1.52	1.82	0.95
4	selective	0	292	559	9793	1.46	1.83	0.92
4	selective	16	527	580	9664	1.48	1.83	0.92
4	selective	32	695	914	9022	1.58	1.84	0.92
4	selective	64	908	1106	9177	1.56	1.85	0.92

Table I. Speculative RAW dependency resolution results

	$\mathrm{W_q} = 1 - \mathrm{W_{hist}}$				Testcase
$\mathrm{W_{spec}}$	0.0	0.5	1.0	Simple	
0	0.0	9.3	9.3	0.0	gf_multiply
2	13.7	12.3	11.0	11.0	
4	13.7	12.3	11.0	11.0	
6	13.7	13.7	13.7	13.7	
0	25.5	25.5	13.0	0.0	priority_mux
2	12.5	12.5	13.2	13.2	
4	12.5	12.5	13.2	13.2	
6	13.2	13.2	13.2	13.2	
0	0.3	1.9	2.0	0.0	median_filter
2	0.3	1.9	2.0	0.0	
4	0.3	1.9	2.0	0.0	
6	0.3	1.9	2.0	0.0	

Table II. Runtime speed-up in % over static prioritization

dynamically computing $P_{qh}(r)$ is efficiently implemented by a two-dimensional, statically pre-computed 64-entry look-up table indexed by $F_q(r)$ and the 2b output value of the GAg predictor of r (selecting the branch probability P_{hist}). At run-time, only the evaluation of $S(r)$ and the shift have to be performed. For computing $P_{simple}(r)$, we set $P_{max}[5{:}0] = 63$. Note that with these parameters, choosing $W_{spec} = 6$ implies that the priority of speculative accesses is always zero.

gf_multiply performs Galois field multiplication. Due to the address of the next read being dependent on the result of a previous read, it has a structure similar to that of Figure 4.b. Thus, the Value-Speculation Prioritization scheme should be profitable here. This is indeed the case: Queue-balancing and control-speculation have only limited speed-up potential, but heavily penalizing speculative accesses over non-speculative ones by increasing W_{spec} achieves speed-ups of up to 13.7% (even when using the simpler P_{simple} dynamic priority).

median_filter performs image processing and executes parallel memory reads on the required pixels. It has no data dependencies between reads and no control flow in the loop body. However, all pixels read are combined in the median computation. The application thus has a structure similar to that shown in Figure 4.c and suggests the use of Queue-Balancing Priority. As expected, the best speed-ups of 2% are achieved with $W_q = 1.0$, the other parameters have no impact.

priority_mux is a synthetic test case for reads on separate branches of a conditional that are executed in parallel on the RCU, reflecting the structure of Figure 4.d. This example profits by up to 25.5% from Control-Speculation Priority by setting $W_q = 0.0 \Leftrightarrow W_{hist} = 1.0$, and disabling the unnecessary Value-Speculation Priority by $W_{spec} = 0$.

These results also show that, if FPGA area is scarce, the simpler P_{simple} dynamic priority scheme might be a viable choice. It does not not need to track queue fill-levels or require any kind of per-port branch prediction, but can still yield performance improvements over the purely static case as long as speculative accesses are at least somewhat penalized over non-speculative ones (e.g., by $W_{spec} = 2$). But even better performance is achievable by matching (ideally, compiler-assisted) the access predictor weights to the application.

VIII. CONCLUSION AND FUTURE WORK

With our prior work on PreCore, we have introduced an effective general-purpose scheme for adding hardware-supported speculation to statically scheduled datapaths compiled from C. The RAP extensions presented here improve PreCoRe by allowing efficient handling of RAW memory dependencies and reducing the negative effects of speculation-induced increased memory pressure by a dynamic prioritization scheme. Using our new Selective Replay method to react to speculative RAW violations, we improve performance over conventional approaches that just rely on victim caches to ensure quick re-execution.

The key capability of reconfigurable computing, the adaptation of the microarchitecture of the generated hardware to the structure of the application, is further strengthened using RAP. In future work, we will refine our compiler to automatically support the required customization of individual RAP features by appropriate heuristics.

Acknowledgments: This work was supported by the German national research foundation DFG and by Xilinx Inc.

REFERENCES

[1] AutoESL. AutoPilot.
[2] T. J. Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*. Ph.D. thesis, University of California, Berkeley, 2002.
[3] H. Gädke, F. Stock et al. Memory Access Parallelisation in High-Level Language Compilation for Reconfigurable Adaptive Computers. In *FPL*, pp. 403–408. 2008.
[4] H. Gädke-Lütjens, B. Thielmann et al. A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation. *Intl. Conf. on Field Programmable Logic and Applications*, 0:475–482, 2010.
[5] 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.
[6] 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. 1999.
[7] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2002.
[8] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, 1990.
[9] D. Kaeli and P.-C. Yew. *Speculative Execution In High Performance Computer Architectures*. CRC Press, Inc., 2005.
[10] N. Kasprzyk and A. Koch. High-Level-Language Compilation for Reconfigurable Computers. In *ReCoSoC*. 2005.
[11] H. Lange and A. Koch. Memory Access Schemes for Configurable Processors. In *FPL*, pp. 615–625. 2000.
[12] H. Lange and A. Koch. Architectures and Execution Models for Hardware/Software Compilation and their System-Level Realization. *IEEE Trans. on Computers*, 2010.
[13] H. Lange, T. Wink et al. MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers. In *DATE*. 2011.
[14] Y. Li, T. Callahan et al. Hardware-software co-design of embedded reconfigurable architectures. In *DAC*, pp. 507–512. 2000.
[15] M. H. Lipasti, C. B. Wilkerson et al. Value Locality and Load Value Prediction. In *Proc. of the 7th intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-VII, pp. 138–147. ACM, New York, NY, USA, 1996.
[16] Mentor Graphics. CatapultC.
[17] Synopsys. Synphony C Compiler.
[18] B. Thielmann, J. Huthmann et al. Evaluation of Speculative Execution Techniques For High-Level Language to Hardware Compilation. In *Proc. of the Reconfigurable Communication-centric SoC ReCoSoC 2011*. Montpellier (F), 2011.
[19] B. Thielmann, J. Huthmann et al. PreCoRe - A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation. In *Proc. Intl. Conf. Field-Programmable Logic and Applications (FPL)*. 2011.
[20] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proc. of the 19th annual intl. symp. on Computer Architecture*, pp. 124–134. 1992.