Widening the Memory Bottleneck by Automatically-Compiled Application-Specific Speculation Mechanisms

Benjamin Thielmann and Jens Huthmann and Thorsten Wink and Andreas Koch

1 Introduction

The rate of improvement in the single-thread performance of conventional central processing units (CPUs) has decreased significantly over the last decade. This is mainly due to the difficulties in obtaining higher clock frequencies. As a consequence, the focus of development has shifted to multi-threaded execution models and multi-core CPU designs instead. Unfortunately, there are still many important algorithms and applications that cannot easily be rewritten to take advantage of this new computing paradigm. Thus, the performance gap between parallelizable algorithms and those depending on single-thread performance has widened significantly. Application-specific hardware accelerators with optimized pipelines are able to provide improved single-thread performance, but have only limited flexibility and require high development effort compared to programming software-programmable processors (SPP).

Adaptive computing systems (ACS) combine the high flexibility of SPPs with the computational power of a reconfigurable hardware accelerator (e.g., using field-programmable gate arrays, FPGA). While ACSs offer a promising alternative compute platform, the compute-intense parts of the applications, the so-called kernels, need to be transformed to hardware implementations, which can then be executed on the reconfigurable compute unit (RCU). Not only performance, but also better usability are key drivers for a broad user acceptance and thus crucial for the practical success of ACSs. To this end, research for the past decade has focused not only on ACS architecture, but also on the development of appropriate tools which enhance the usability of adaptive computers. The aim of many of these projects is to create hardware descriptions for application-specific hardware accelerators automatically from high-level languages (HLL) such as C.

Benjamin Thielmann · Jens Huthmann · Thorsten Wink · Andreas Koch
Embedded Systems and Applications Group, Technische Universität Darmstadt, Germany
e-mail: {thielmann,huthmann,wink,koch}@esa.cs.tu-darmstadt.de

To achieve high performance, the parallelism inherent to the application needs to be extracted and mapped to parallel hardware structures. Since the extraction of coarse-grain parallelism (task/thread-level) from sequential programs is still a largely unsolved problem, most practical approaches concentrate on exploiting instruction-level parallelism (ILP). However, ILP-based speed-ups are often limited by the memory bottleneck. Commonly, only 20% of the instructions of a program are memory accesses, but they require up to 100x the execution time of the register-based operations [13]. Furthermore, memory data dependencies also limit the degree of ILP from tens to (at the most) hundreds of instructions, even if support for unlimited ILP in hardware is assumed [10].

For this reason, memory accesses need to be issued, processed, and dependencies resolved as quickly as possible. Many proposed architectures for RCUs rely on local low-latency high bandwidth on-chip memories to achieve this. While these local memories have become more common in modern FPGA devices, their total capacity is still insufficient for many applications and low-latency access to large off-chip memory remains necessary for many applications.

As another measure to widen the memory bottleneck for higher ILP, *speculative* memory accesses can be employed [6]. We use the general term "speculative" to encompass uncertain values (has the correct value been delivered?), control flow (has the correct branch of a conditional been selected and is the access needed in this branch?), and data dependency speculation (have data dependencies been resolved?). To efficiently deal with these uncertainties (e.g., by keeping track of speculative data and resolving data dependencies as they occur), hardware support in the compute units is required. We will describe an approach that efficiently generates these hardware support structures in an application-specific manner from a high-level description (C program), instead of attempting to extend the RCU with a general-purpose speculation block. To this end, we will present the speculation-handling micro-architecture PreCoRe, the high-level language hardware compile flow Nymble, and the back-end memory system MARC II, which was tuned to support the speculation mechanisms.

2 Overview

The development of a compiler and an appropriate architecture is a highly interdependent task. Most of the high-level language to hardware compilers developed so far use static scheduling for their generated hardware datapaths. A major drawback of this approach is its handling of variable-latency operators, which forces a statically scheduled datapath to completely stall all operations on the accelerator until the delayed operation completes. Such a scenario is likely to occur when accessing cached memories and the requested data cannot be delivered immediately. Dynamic scheduling can overcome this issue, but has drawbacks such as its complex execution model, which results in considerable hardware overhead and lower clock rates.

Furthermore, in itself, it does not address the memory bottleneck imposed by the high latencies and low bandwidth of external memory accesses.

Due to these limitations, RCUs are becoming affected by the processor/memory performance gap that has been plaguing CPUs for years [10]. But since RCU performance depends heavily on exploiting parallelism with hundreds of parallel operators, RCUs suffer a more severe performance degradation than CPUs, which generally have only few parallel execution units in a single core.

The quest for high parallelism in ACSs further emphasizes this issue. Control flow parallelism allows to simultaneously execute alternative precluding branches, such as those in an if/else construct, even before the respective control condition has been resolved. However, such an exploitation of parallel control flows may cause additional memory traffic. In the end, this can even slow down execution over simpler less parallel approaches.

A direct attempt to address the negative effect of long memory access latencies and insufficient memory bandwidth is the development of a sophisticated multi-port memory access system with distributed caches, possibly supported by multiple parallel channels to main memory [17]. Such a system performs best if many *independent* memory accesses are present in the program. Otherwise, the associated coherency traffic would become a new bottleneck. Even though this approach helps to benefit from the available memory bandwidth and often reduces access latencies, stalling is still required whenever a memory access cannot be served directly from one of the distributed caches.

Load value speculation is a well-studied, but rarely used technique to reduce the impact of the memory bottleneck [19]. Mock et al. were able to prove by means of a modified C compiler, which forced data speculation on an Intel Itanium 2 CPU architecture where possible, that performance increases due to load value speculation [22] of up to 10% were achievable. On the other hand, the Itanium 2 roll-back mechanism, which is based on the Advanced Load Address Table (ALAT), a dedicated hardware structure that usually needs to be explicitly controlled by the programmer [20], produces performance losses of up to 5% under adverse conditions with frequent misspeculations.

Research on data speculation methods and their accuracy has produced a broad variety of data predictors. History based predictors select one of the previously loaded values as the next value, solely based on their occurrence probability. Stride predictors do not store absolute values, but determine the offset between the successive loaded values. Here, instead of an absolute value, the most likely *offset* is selected. In this manner, sequences with constant offset between elements can be predicted accurately. Both techniques have proven to be beneficial and do not require long learning time, but both fail to provide good results for complex data sequences. Thus, more advanced techniques, such as context-based value predictors, predict values or strides as the function of a previously observed data *sequence* [24]. Performance gains are achievable if the successful prediction rate is high, or if the penalty to recover from misspeculations is very low.

The load value speculation technique is especially beneficial for statically scheduled hardware units, since now even the variable-latency cached read operations

give the appearance of completing in constant time (by returning a speculated value on cache misses). This allows subsequent operations to continue to compute speculatively, instead of stalling non-productively. As the predicted values may turn out to be incorrect, the microarchitecture must be extended to re-execute the affected parts of the computation with correct operands (replayed), and commit only those results computed from values that were either correctly speculated or actually retrieved from memory. In this approach, memory reads are the sole source of speculative data, but intermediate computations may be affected by multiple reads. Even a correct speculation might be poisoned by a later incorrectly speculated read value. Ideally, only those computations actually affected by the misspeculated value need to be replayed. While this could be handled at the granularity of individual operators, it would require complex control logic similar to that of dynamically scheduled hardware units. As an alternative, our proposed approach will manage speculation on groups of operators organized as *Stages*, which are similar to the start cycles in a static schedule.

It is important to note that by continuing execution speculatively, an increased number of memory read accesses is issued and then possibly replayed onece or several times, increasing the pressure on the memory system even more. Additionally, data dependency violations are likely to occur in such an out-of-order execution of accesses and also need to be managed. We propose prioritization and data dependency resolution schemes to address these issues a run-time.

The speculation support mechanisms, collectively named PreCoRe, are lightweight extensions to a statically scheduled datapath; they do not require the full flexibility (and corresponding overhead) of datapaths dynamically scheduled at the level of individual operators. PreCoRe focuses on avoiding slow-downs of the computation compared to a non-speculative version (by not requiring additional clock cycles due to speculation overhead), even if all speculations would fail continuously. The PreCoRe microarchitecture extensions are automatically generated in an application-specific manner using the Nymble C-to-hardware compiler. At run-time, they rely on the MARC II memory subsystem to support parallel memory accesses and handle coherency issues. Together, these components provide an integrated solution to enable efficient speculative execution in adaptive computing systems.

3 The PreCoRe Speculation Framework

PreCoRe (predict, commit, replay) is a new execution paradigm for introducing load value speculation into statically scheduled data paths: Load values are predicted on cache misses to hide the access latency on each memory read request. Once the true value has actually been retrieved from the memory, one of two operations must happen: If the previous speculatively issued value matches the actual memory data, PreCoRe commits all dependent computations which have been performed using the speculative value in the meantime as being correct. Otherwise, PreCoRe reverts

those computations by eliminating speculatively generated data and issuing a replay of the affected operations with corrected values. To implement the PreCoRe operations, three key mechanisms are required. First, a load value speculation unit is needed to generate speculative data for each memory read access within a single clock cycle. Second, all computations are tagged with tokens indicating their speculation state. The token mechanism is also used to commit correct or to eliminate speculative computations. Third, specialized queues are required to buffer intermediate values, both before they are being processed as well as to keep them available for eventual replays. All three key mechanism will be introduced and discussed in this section.

3.1 Load Value Speculation

Evidently the benefit achieved by speculation is highly dependent on the accuracy of the load value prediction. Fortunately, data speculation techniques have been well explored in the context of conventional processors [28, 3].

It is not possible in the spatial computing paradigm (with many distributed loads and stores) to efficiently realize a predictor with a global perspective of the execution context. This is the opposite of processor-centric approaches, which generally have very few Load Store Units (LSU) that are easily considered globally. On an RCU, the value predictors have a purely local (per port) view of the load value streams. This limited scope will have both detrimental and beneficial effects: On one hand, a predictor requires more training to accumulate enough experience from its own local data stream to make accurate prediction. On the other hand, predictors will be more resilient against irregular data patterns (which would lead to deteriorated accuracy) flowing through other memory ports.

Using value speculation raises the question of how to train the predictors, specifically, when the underlying pattern database (on which future predictions are based) should be updated: Solely if a speculation has already been determined as being correct/incorrect? Since this could entail actually waiting for the read of main memory, it might take considerable time. Or should the speculated values be assumed to be correct (and entered into the pattern database) until proven incorrect later? The latter option was chosen for the PreCoRe, because a single inaccurate prediction will always lead to the re-execution of all later read operations, now with pattern databases updated with the correct values. The difference to the former approach is that the predictor hardware needs to be able to rollback the entire pattern database (and not just individual entries) to the last completely correct state once a speculation has proven to be incorrect. One of the overarching goals of PreCoRe remains to support these operations without slowing down the datapaths over their non-speculative versions (see Section 6).

3.1.1 Predictor Architecture

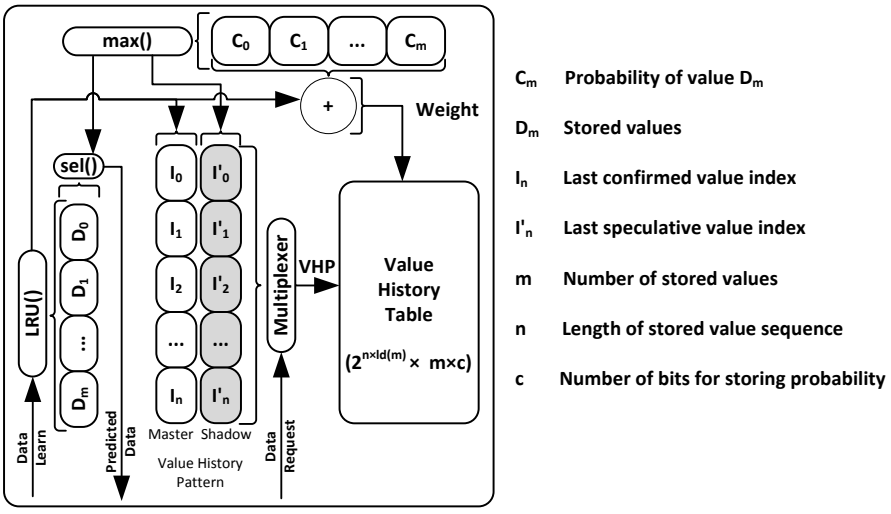

Fig. 1 Local history-based load value predictor

The value predictors (shown in Figure 1) follow a two-level finite-context scheme, an approach that was initially used in branch prediction. The predictions exploit a correlation of a stored history of prior data values to derive future values [28]. The precise nature of the correlation is flexibly parametrized: The same base architecture is used to realize both last value prediction (which predicts a future value by selecting it from a set of previously observed values, e.g., *23-7-42-23-7-42*) and stride prediction (which extrapolates a new value from a sequence of previously known strides, e.g., from the strides *4-4-8-4*, the sequence *0-4-8-16-20-24-28-36-40* is predicted). A PreCoRe value prediction unit operates parallel last-value and stride sub-predictors in tournament mode, where a sub-predictor is trusted until it mispredicts, leading to a switch to the other sub-predictor. Since both sub-predictors use the same micro-architecture (with exception of the correlation computation), we will focus the discussion on just one mode, namely the value-speculation.

The predictor not only keeps track of the last m different values $D_1, \ldots, D_m$ in a least-recently-used fashion in its pattern database D, but also maintains the n-element sequence $I_1, \ldots, I_n$ in which these values occurred (the Value History Pattern, VHP). Each of the n elements of I is an $\lceil \log_2 m \rceil$-bit wide field holding an index reference to an actual value stored in D. I is used in its entirety to index the Value History Table (VHT) to determine the most likely of the currently known values: Each entry in the VHT expresses the likelihood for all of the known values D_i as a c-bit unsigned counter C_i, with the highest counter indicating the most likely value (on ties, the smallest i wins). The VHT is thus accessed by a $n \cdot \lceil \log_2 m \rceil$-bit wide

address and stores $m \cdot c$ bits wide words. On start-up, each VHT counter is initialized to the value 2^{c-1}, indicating a value probability of $\approx 50\%$.

To handle mispredictions, we keep two copies of the VHP as I and I': I is the master VHP, which stores *only* values that were already confirmed as being correct by the memory system. However, the stored values may be *outdated* with respect to the actual execution (since it might take awhile for the memory system to confirm/refute the correctness of a value). The shadow VHP I' (shown with gray background in the figure) additionally includes speculated values of *unknown* correctness. It accurately reflects the *current* progress of the execution. Values will be predicted based on the shadow VHP until a misprediction is discovered. The computation in the datapath will then be replayed using the last values not already proven incorrect. A similar effect is achieved in the predictor by copying the master VHP I (holding correct values) to the shadow VHP I' (basing the next predictions on the corrected values). [25] explains the predictor in greater detail and shows a step-by-step example of its operations.

The predictor is characterized by the two parameters n and m. The first is the maximum length of the context sequence, the second the maximum number of different values tracked. The state size of VHT and VHP (and thus the learning time before accurate predictions can be made) grows linearly in n and logarithmically in m. Note that in a later refinement, optimum values for n and m could be derived by the compiler using profile-guided optimization methods.

3.2 Token Handling Mechanisms

The PreCoRe mechanisms are inserted into the datapath and controller of a statically scheduled hardware unit. They are intended to be automatically created in an application-specific manner by the hardware compiler. With the extensions, cache-misses on reads no longer halt execution due to violated static latency expectations, but allow the computation to proceed using speculated values. Variable latency reads thus give the appearance of being fixed-latency operators that always produce/accept data after a single cycle (as in cache-hit case).

In this manner, predicted or speculatively computed values propagate in the datapath. However, only reversible (side effect-free) operations may be performed speculatively to allow replay in case of a misprediction. In our system, write operations thus form a *speculation boundary*: A write may only execute with operand values that have been confirmed as being correct. If such a confirmation is still absent, the write will stall until the confirmation arrives. Should the memory system refute the speculated values, the entire computation leading up to the write will be replayed with the correct data.

This is outlined in the example of Figure 2.a. Here, the system has to ensure that the data to be written has been correctly predicted in its originating READ node (the sole source of speculated data in the current PreCoRe prototype) before the WRITE node is allowed to execute. This is achieved for the READ by comparing

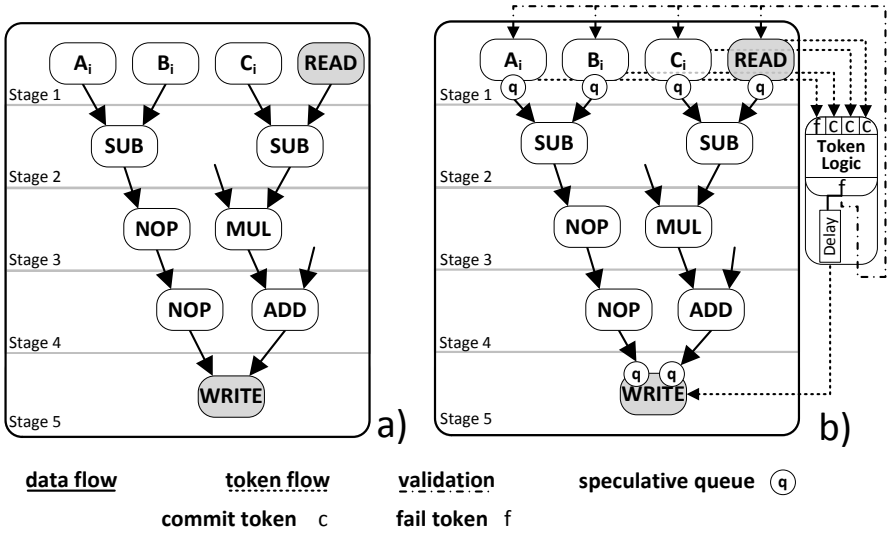

Fig. 2 Datapath and speculation token processing

the predicted read result, which is retained for this purpose in an output queue in the READ node, with the actual value received later from the memory system. Until the comparison has established the correctness of the predicted value, the data to be written (which was computed depending on the predicted read value) is held in an input queue at the WRITE node. This queue also gives the WRITE node the appearance of a single-cycle operation, even on a cache-miss.

Figure 2.b sketches the extension of the initial statically scheduled datapath with PreCoRe: Explicit tokens track the speculativity of values and their confirmation/refutation events. This is indicated by additional edges that show the flow of tokens and validation signals.

As an example, if the READ node has confirmed a match between predicted and actual data values, it indicates this by sending a Commit-Token (shown as C in the Figure) to the Token Logic. However, to reduce the hardware complexity, this token is *not* directly forwarded to the WRITE node waiting for this confirmation, as would be done in operator-level speculation. Instead, the speculativity is tracked per data path Stage (corresponding to the operators starting in the same clock cycle in a static schedule). Only if *all* operators in a Stage confirm their outputs as being correct, is the C-Token actually forwarded to the WRITE operator acting as speculation boundary, confirming as correct the oldest WRITE operand with uncertain speculation status. Speculated values and their corresponding C- and F-Tokens (indicating failed speculation) always remain in order. Thus, no additional administrative information, such as transaction IDs or similar, is required. Tokens are allowed to temporarily overtake their associated data values up to the next synchronization point (see Section 3.3) by skipping Stages that lack speculative operators (READ nodes). The speculation output status of a Stage depends on that of its inputs: It will

be non-speculative, if no speculative values were input, and speculative, if even a single input to the Stage was speculative. In the example, Stages 2...4 do not contain READs, the C-Token can thus be directly forwarded to the WRITE in Stage 5, where it will be held in a token queue until the correctly speculated value arrives and allows the WRITE to proceed. In parallel to this, pipelining will have lead to the generation of more speculative values in the READ, which continue to flow into the subsequent Stages.

If the initial speculation in the READ node failed (the output value was discovered to be mispredicted), all data values which depended on the misspeculated value have to be deleted, and the affected computations have to be replayed with the correct no-longer speculative result of the READ. This is achieved by the Token Logic recognizing that the misspeculated READ belonged to Stage 1, and thus the entire Stage is considered to have misspeculated. All Stages relying on operands from Stage will be replayed. The F-Token does not take effect immediately (as the C-Token did), but is delayed by the number of Stages between the the speculated READ and the WRITE at the speculation boundary. In the example, the F-Token will be delayed by three Stages, equivalent to three clock cycles of the datapath actually computing. If the datapath 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. Delaying the effect of the F-Token ensures that the intermediate values computed using the misspeculated value in Stages 2 ... 4 have actually arrived in the input queues of the WRITE operation in Stage 5, and will be held there since no corresponding C- or F-Token for them was received earlier. At this time, the delayed F-Token arrives at the WRITE and deletes three sets (corresponding to the three intermediate Stages) of potentially incorrect input operands from the WRITE input queues and thus prevents it from executing. The replay of the intermediate computation starts immediately once the last attempt has been discovered to have used misspeculated values. Together with the correct value from the READ (retrieved from memory), the other nodes in Stage 1 re-output their last results (which may still be speculative themselves!) from their output queues and perform the computations in Stages 2...4 again. A more detailed example of token handling is given in [26].

3.3 Queue Management for Speculation

First introduced in the previous Section, operator output queues (re-)supply the data to allow replay operations and are thus essential components of the PreCoRe architecture. Note that some or all of the supplied values may be speculative Data values are retained until all outputs of a stage have been confirmed and a replay using these values will no longer be required. Internally, each queue consists of separate sub-queues for data values and tokens, with the individual values and tokens being associated by remaining strictly in order: Even though tokens may overtake data values between Stages, their sequence will not be changed. In our initial description,

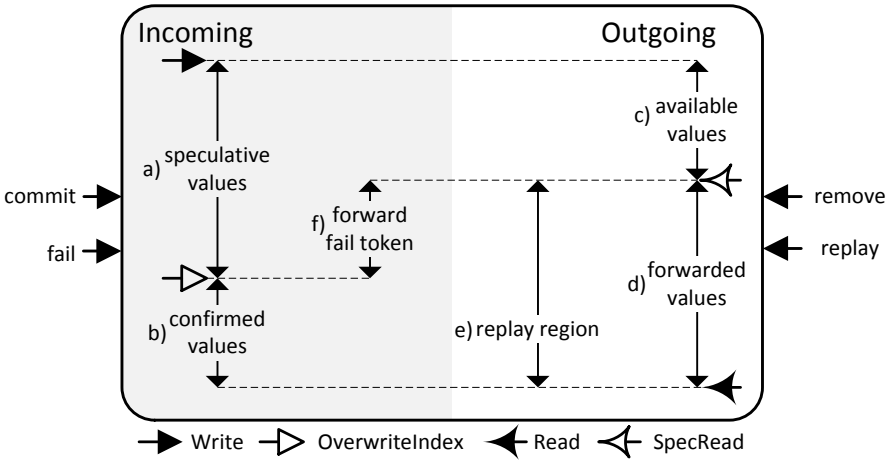

Fig. 3 Value regions in speculative queue

we will concentrate on the more complex output queues. Input queues are simpler and will be discussed afterwards.

Figure 3 gives an overview of an output queue, looking at it from the incoming (left side) and outgoing (right side) perspectives.

On the incoming side, values are separated into two regions: speculative values (a) and confirmed values (b). Since all data values are committed sequentially and no more committed data may arrive once the first speculative data entered the queue, these regions are contiguous. Similarly, outgoing values are in different contiguous regions depending on their state: (d) is the region of values that have already been forwarded as operands to a consumer node and are just retained for possible replays, (c) are the values that are available for forwarding. Conventional queue behavior is realized using the Write pointer to insert newly incoming speculative data at the start of region (a), and the Read pointer to remove a value from the end of region (d) after the entire Stage has been confirmed. Two additional pointers are required to implement the extra regions: Looking into the queue from the outgoing end, SpecRead determines the first value which has *not* been forwarded yet at the end of region (c), and OverwriteIndex points to the last confirmed value at the beginning of region (b).

On misspeculation in a predecessor Stage (indicated by an incoming F-Token), the speculative values making up region (a) are discarded by setting Write back to OverwriteIndex. If the queue does not hold confirmed values (regions (c) and (d) are empty), the F-Token is passed on (f) through the output queue into subsequent Stages.

Confirmed values are forwarded to the consumer nodes but retained in the queue for replays (moving from region (c) into region (d) by manipulation of the SpecRead pointer). If operators in the same Stage request a replay, SpecRead is reset to Read, making all of the already forwarded but retained values available again for re-execution of subsequent stages, with (f) now acting as a replay region (e). Retained

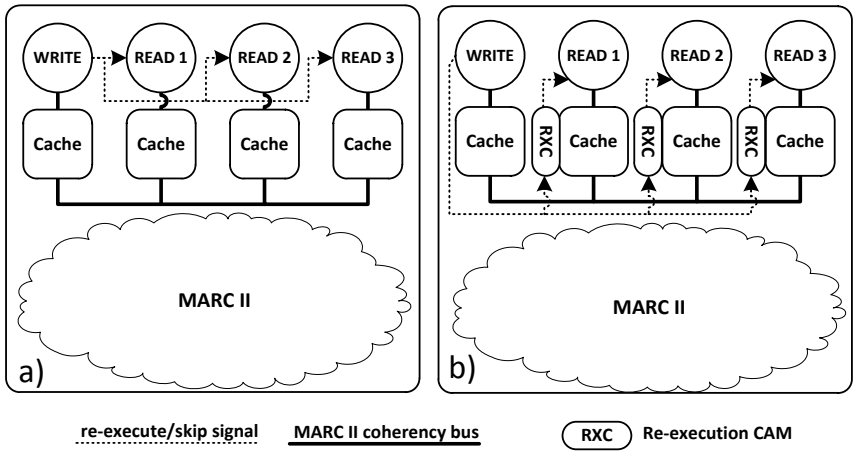

Fig. 4 Resolution schemes for RAW memory dependencies

values are removed from the queue *only* if all operators in the Stage have confirmed their execution (and thus ruled out the need for a future replay). This final removal is achieved using the Read pointer. For a detailed example of the output queue operation, please refer to [26].

Input queues have a similar behavior, but do not need to confirm speculative data (that was handled in their predecessor's output queue).

3.4 Dynamic Resolution of RAW Dependencies

The speculative PreCoRe execution scheme enables the prefetching of memory reads: A read which might originally be scheduled after a write, is allowed to execute speculatively before the write has finished. This reordering potentially violates a read-after-write (RAW) memory data dependency. Thus, all of the memory read accesses potentially depending on the write must remain in speculative state until the memory write access itself has been committed. Static points-to/alias analysis in the compiler can remove some of the potential dependencies and guarantee that reads and writes will be to non-overlapping memory regions (allowing out-of-order prefetching). However, in most realistic cases, such guarantees cannot be assured at compile time. Instead, dynamic detection and correction of dependency violation due to speculatively prefetched reads must be employed to handle the general case. PreCoRe supports two such mechanisms.

Universal Replay: This approach is a straightforward, low area, but suboptimal extension of the existing PreCoRe commit/replay mechanisms: *All* RAW dependency-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 PreCoRe, the speculation depth is determined by the length of speculation value queues on the Stages between the possibly dependent read and write nodes.

In practice, Universal Replay is less inefficient as it appears at first glance: Assuming that the data written by all write operations is still present in the cache, the replays will be very quick. Also, in the scheme, *all* writes are initially assumed to induce a RAW violation. If a write is only conditionally executed, the potentially dependent reads can be informed if the evaluation of the control condition prevents the write from executing at all. This is communicated from each write to the reads using a Skip signal (see Figure 4.a). If all writes have been skipped, there no longer is a risk of a RAW violation, and the data retrieved by the reads will be correct (and can be confirmed as such). On the other hand, the replays in this scheme can become expensive if the write data has been displaced from the cache, or if the replayed computation itself is very complex. Thus, it is worthwhile to examine a better dependency resolution scheme.

Selective Replay: This more refined technique avoids unnecessary replays by actually detecting individual read/write address overlaps on a per-port basis and replays only those RAW-speculated reads that were actually affected by writes. To this end, read ports in the memory subsystem are extended with dedicated hardware structures (RXC, see Section 5.3) to detect and signal RAW violations. Combined with the Skip signal to ignore writes skipped due to control flow, replays are only started for specific read nodes if RAW violations did actually occur.

3.5 Access Prioritization

PreCoRe fully exploits the spatial computing paradigm by managing operations on the independent parallel memory ports supplied by the MARC II memory subsystem (see Section 5). However, internally to MARC II, time-multiplexed access to shared resources, such as buses or the external memory itself, becomes necessary. By carefully prioritizing different kinds of accesses, the negative performance impact of such time-multiplexing can be reduced. PreCoRe influences these priorities not only to best use the available bandwidth on the shared resources for useful accesses, but also to employ spare bandwidth to perform prefetching. A number of techniques is used to manage access priorities.

The simplest approach consists of statically allocating the priorities at compile time. In PreCoRe, the write port always executes with the highest priority, since it will only be fed with non-speculative data and will thus always be useful. Read operations placed early in the static schedule will be assigned a higher priority than read operations scheduled later, so their data will already be available when later stages execute. In Figure 5.a, READ1 thus executes with higher priority than READ2.

Figure 5.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

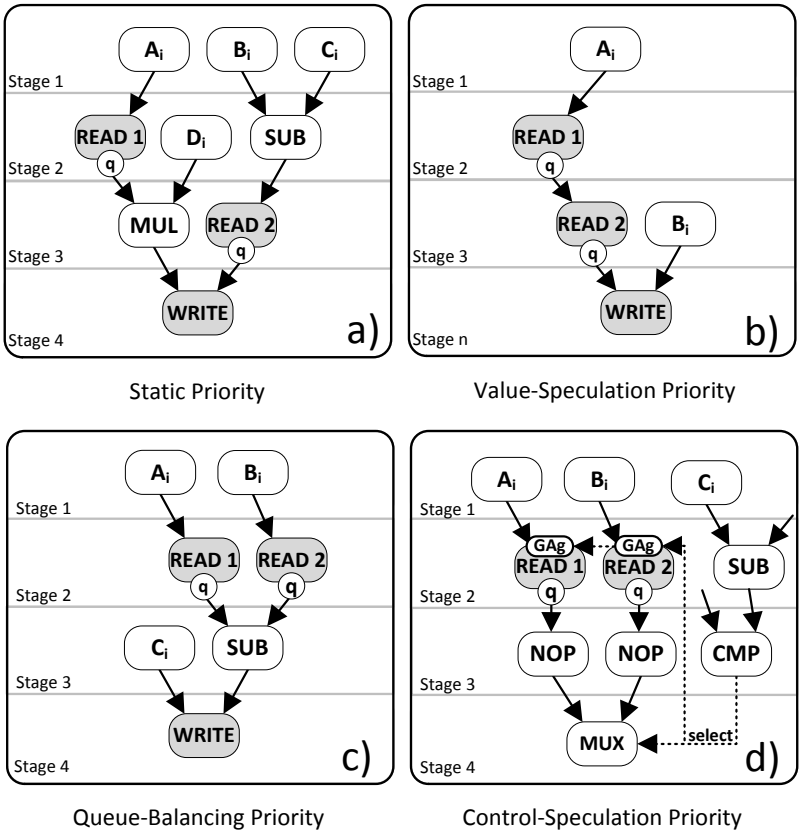

Fig. 5 Scenarios for priority-based shared resource arbitration

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). These resources would have been better used to execute the non-address speculated READ1 of the *next* loop iteration, which is an access that will always be useful. Value-speculation priority dynamically lowers the priority of accesses operating on speculated addresses and/or data values, thus giving preferential treatment to accesses using known-correct operands.

In some situations, the simple static per-port priority can even lead to a loss of performance. This occurs specifically if the outputs of multiple reads at the same stage converge at a later operator. An example for this is shown in Figure 5.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 number, 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.

As described above, performance gains may be achieved by allowing read operators to immediately reply with a speculated data value on a cache-miss. Orthogonal to this data speculation approach is speculating on whether to execute the read operator at all. Such *control-speculation* is performed on software-programmable processors using branch prediction techniques. While this approach is not directly 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/buses: For software, branch prediction would execute only the most likely used read in a conditional, while the RCU would attempt to execute the reads on all branches of the conditional in parallel, leading to heavy competition for the shared resources and potentially slowing down the overall execution (on multiple parallel cache misses).

To alleviate the problem, we track which branch of a parallel conditional actually performed useful computations by recording the evaluated control condition. The read operators in that branch will receive higher priorities, thus preventing reads in less frequently-taken branches from hogging shared resources. To this end, we use decision tracking mechanisms well established in branch prediction, specifically the GAg scheme [29], but add these to the individual read operators of the parallel conditional branches (see Figure 5.d). The trackers are connected to the controlling condition for each branch (see [17] for details) and can thus attempt to predict which branch of the past branching history will be useful next, 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.

To exploit the advantages of the different schemes, they are all combined into a general dynamic priority computation.

$$\mathsf{P}_{\mathsf{dyn}}(r) = (\mathsf{W}_{\mathsf{q}} \cdot \mathsf{P}_{\mathsf{q}}(r) + (1 - \mathsf{W}_{\mathsf{q}}) \cdot \mathsf{P}_{\mathsf{hist}}(r)) \cdot 2^{-(\mathsf{W}_{\mathsf{spec}} \cdot \mathsf{IsSpec}(r))}$$

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 $= 1$ 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), and $= 0$ otherwise. This predicate will be used to penalize the priority of speculative accesses. 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. See [27] for a more detailed discussion and an evaluation of the performance impact of these parameters.

4 The Nymble C-to-Hardware Compiler

To discuss the integration of PreCoRe into the Nymble compile flow, we will first give an overview of the initial C-to-hardware compilation process. It relies on classical high-level synthesis techniques to create synthesizable RTL descriptions of the resulting statically scheduled hardware units.

4.1 Control-Data Flow Graph Generation

We use a simple program computing the factorial function (Listing 1) as running example for the compilation flow.

Listing 1 Sample program for hardware compilation

```
int factorial(int n) {
  int a = 1;
  int i;

  for(i=1; i<n; ++i}
    a = a * i;

  return a;
}
```

The Nymble front-end relies on traditional compiler techniques (specifically, those of the Scale framework [9] [7]) to lex and parse the input source code, perform machine-independent optimizations, finally representing the program as Control Flow Graph (CFG) in Static Single Assignment (SSA) form [1]. Figure 6 shows the SSA-CFG of our sample program. SSA-CFGs are a commonly used intermediate representation in modern software compilers and well suited for the actual hardware compilation.

In SSA form, each variable is written only once, but may be read multiple times. Multiple assignments to the same variable create new value instances (versions of the variable, often indicated by subscripts). A Φ function selects the current value instance when multiple value instances converge at a CFG node. This happens, e.g., for conditionals for the true and false branches, or for the entering and back-edge of a loop.

Given the abundance of flip-flops on most current reconfigurable devices, the value instances of the SSA form could be mapped directly to hardware registers (but also see comment below). To build the computations between the registers, the data-flow from source variables through operators to destination variables has to be extracted. This is easily achievable in SSA form, since a value can originate only

from one specific assignment. The flow of values is expressed as a Data Flow Graph (DFG), shown in Figure 7 for the example.

While it would suffice for the synthesis of the datapath of the hardware unit (by mapping the operators to compute nodes and the edges to appropriate wiring), the control flow (e.g., the loop termination condition) must still be considered when synthesizing the controller. This is achieved by extending the DFG with control edges (shown as dotted lines in Figure 8, labeled on which boolean value of the controlling condition they activate). Control edges carry the boolean results of conditions to either activate specific nodes (e.g., the end node indicating the completion of hardware execution) or to lect which value instance to pass through the multiplexers representing the Φ functions. For the loops shown here, the Φ functions at the loop heads are controlled by a dedicated init node that outputs true on its control edge if loops are being entered for the first time, and false otherwise.

As a refinement of mapping SSA value instances to registers, it is possible to remove purely intermediate variables and replace them by simple wiring to their computing operator in the DFG, instead of allocating a hardware register to hold the intermediate result.

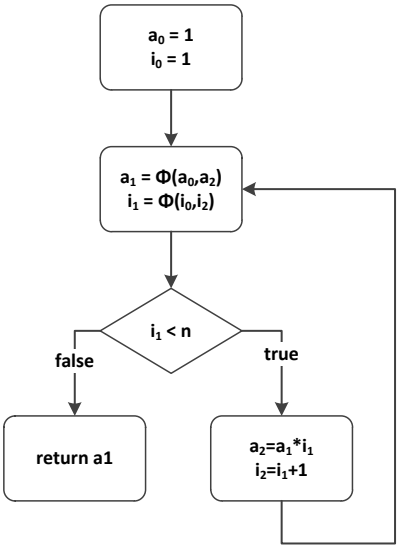

Fig. 6 SSA-CFG of sample program

4.2 Operation Scheduling

An acyclic CDFG could be mapped directly to a purely combinational datapath, evaluating the entire computation in a single clock cycle. However, this approach

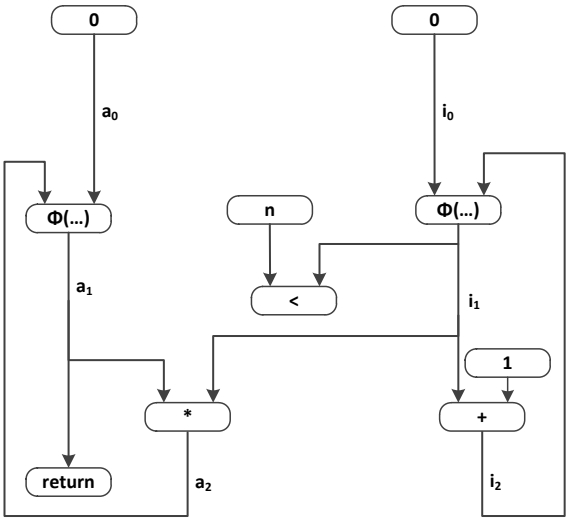

Fig. 7 Data Flow Graph (DFG) of sample program

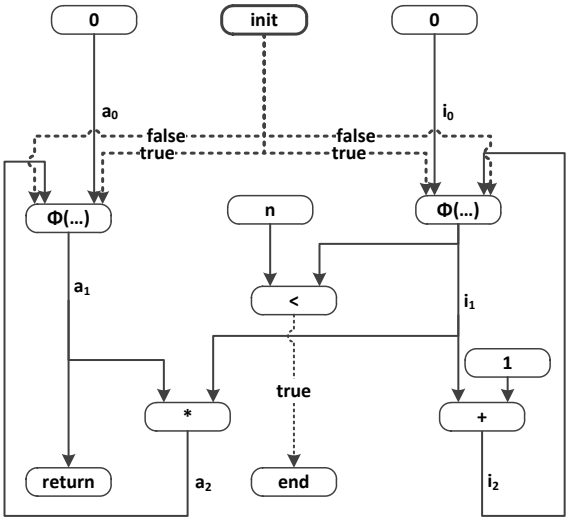

Fig. 8 Control-Data Flow Graph (CDFG) of sample program

would lead to slow clock frequencies and not allow the execution of loops. Thus, the conventional solution is to distribute the computation over multiple clock cycles, leading to both faster clocks as well as allowing cycles (with inserted registers). In a simple implementation of this approach, a hardware registers could be inserted after each operation. Note that further optimizations from high-level hardware synthesis might deviate from this scheme (e.g., packing multiple operators into a clock cycle by operator chaining [21]).

After realizing the computation in sequential logic, the question remains how to control its execution (e.g., when to assert the registers' load inputs to accept newly computed values). This decision is called *scheduling* and can be performed both statically (at compile time) or dynamically (at execution time).

Dynamic scheduling does have numerous advantages: It can easily handle variable-latency operators, such as cached memory accesses, as the decision to store the read value is made only when the read port has indicated that the datum is available. Similarly, conditionals with differing computation times in their true and false branches can also consider the specific path taken at execution time to load the newly computed values at the correct time. Due to these advantages, dynamic scheduling has been used in a number of hardware compilers, such as COMRADE [5], CHiMPS [23], or CASH [2].

On the other hand, the additional logic required to make scheduling decisions at run-time potentially carries a large area overhead, especially when complex control flows have to implemented. In static scheduling, the times when to load newly computed values into registers and when to start new operations are determined at compile time. This is easy for fixed-latency operators, and the case of imbalanced conditional paths can be addressed by padding the shorter path with additional registers to the length of the longer path, equalizing the lengths. However, variable-latency operators pose a significant problem. In practice, they are assumed to execute in a fixed expected latency (e.g., single cycle on a cache hit). Dedicated logic detects at execution time when this assumption does not hold (e.g., on a cache miss), and halts (stalls) the entire datapath until the outstanding datum is actually available. Only then is execution allowed to proceed, giving the rest of the datapath the impression that variable latency operators always provide their results within a fixed time. As an advantage, the control logic for orchestrating the execution of a statically scheduled hardware unit can be implemented in a compact and fast fashion (often just using multi-tapped shift registers). Hardware compilers using static scheduling include GarpCC [4], ROCCC [8], and the base microarchitecture in the Nymble flow.

4.3 Hardware Synthesis in Nymble

With the fundamentals of the hardware synthesis now established, this section will consider some of the details of the Nymble compilation process in greater detail. Nymble actually partitions the SSA-CFG into a hierarchical CDFG, with each loop appearing as a single variable-latency node in the parent CDFG. In this manner, arbitrarily nested loop structures are supported. This is shown in Figure 9: The top level CDFG is the entire factorial function, which accepts a parameter n from software. At this level, the loop has been encapsulated as a single operation. When it detects the loop termination condition, it signals the end of hardware execution to the hardware/software interface layer [16] and passes back the computed factorial from hardware to software.

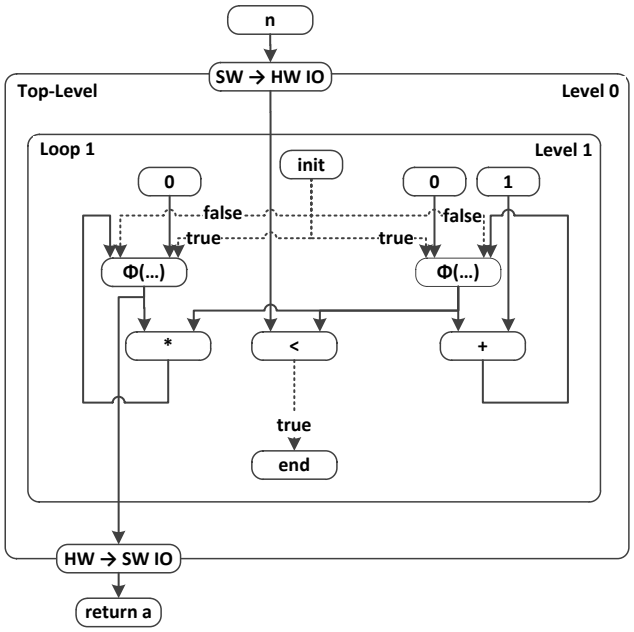

Fig. 9 Hierarchically scheduled CDFG for sample program

Since we compile for the ACS target to a fully spatial hardware implementation with no operator reuse, we can employ a variant of the classical As-Soon-As-Possible (ASAP) static scheduling algorithm [21], adding just minor extensions to obey explicit constraints (discussed in Section 4.4).

Start times of operations are computed from the start times and expected latencies of their predecessor operations. Outer loops are stalled until nested inner loops explicitly signal their completion to the outer loop.

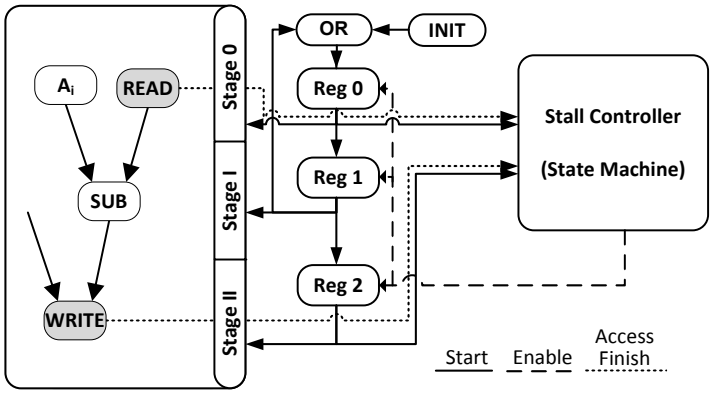

Fig. 10 Synthesized controller for a non-speculative datapath

The hardware controller, sketched in Figure 10, consists of a simple sequencer Reg 0...Reg 2 that just asserts the start signals (if required) of operators scheduled in the same cycle (called a Stage in PreCoRe terminology) and loads the intermediate results of each operator into registers the expected latency number of cycles later. To support pipelining, the sequencer allows multiple stages to be active at the same time. This is limited by backward data dependencies in the DFG, though, which will lead to a longer Initiation Interval (II) between datapath starts. As a second function beyond the sequencing, a Stall Controller also detects violations of expected latency for variable-latency operators and stops the sequencing of all other operations until the variable-latency operator has actually completed. In the base version of Nymble, this applies to nested loops (treated as single operators), and cached memory accesses. The latter will be handled differently with the PreCoRe mechanisms described in the next Section.

4.4 Compiling for the PreCoRe Microarchitecture

PreCoRe requires the extension of the pure statically scheduled execution model of the base version of Nymble to a semi-statically scheduled version that makes more scheduling decisions at execution time, but far fewer than would be made in fully dynamic scheduling. In this section, we will discuss the changes required to the Nymble controller microarchitecture to integrate PreCoRe token handling (Section 3.2) and speculative queues (Section 3.3).

The Stage-based nature of PreCoRe speculation has an impact on the static scheduling of multi-cycle operators in Nymble. In general, such multi-cycle operators will not support a partial replay, especially if they are obtained as third-party IP blocks (e.g., floating-point cores etc.) and will lack the required functionality (injection of preserved state data into the internals of the operator on a replay). Thus, all such operators are constrained in Nymble to be ASAP-scheduled either *completely* before or after any reads (which initiate replays on a misprediction).

Some parts of the controller actually are simplified by using PreCoRe. Since memory reads now become single-cycle operations due to value speculation, the Stall Controller on a read cache miss is no longer required. However, the need to support replays adds extra complexity. The microarchitecture of a controller supporting PreCoRe is sketched in Figure 11, the key changes will be discussed next.

The simple sequencing registers in the original statically scheduled controller are replaced by so-called Flow Control nodes in the PreCoRe controller. During normal execution (no mispredicts), their behavior corresponds to those of the simple shift register controller - the Start signal is delayed by a single clock cycle and passed to the subsequent stage. However, special logic is required to handle replays and to halt further computations in the operation pipeline as soon as a read is discovered to have mispredicted.

The easier of the extensions deals with the management of the input queues in read and write operators: Execution sequencing is only allowed to proceed if *all*

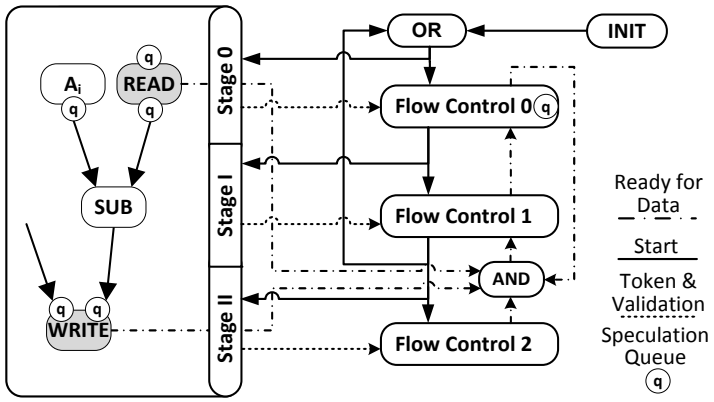

Fig. 11 Synthesized controller for PreCoRe-speculative datapath

input queues in the entire datapath have space (indicated by asserting their Ready for Data signal) for the operands that would be incoming in the next cycle. Lacking such space, sequencing at the datapath level is stopped, but all memory operators are allowed to proceed internally, draining their input queues. Once queue space has become available once more, datapath sequencing continues.

Flow Control nodes of Stages holding speculative operators (such as memory reads) have another extension over the simple sequencing registers: They have internal queues to buffer incoming Start tokens. If their corresponding datapath Stage requests a replay (a read discovered it mispredicted), the start tokens are reissued from the Flow Control token queue to restart the subsequent Stages. If multiple mispredictions occur, the re-issue rate of the replayed start tokens is throttled to match the original Initiation Interval, thus keeping the static parts of the schedule valid. Only once a Stage is confirmed in its entirety (precluding the need for a future replay) is the start token removed from the Flow Control token queue. Analogously to the capacity check for input queues in the datapath, execution in the controller is only allowed to proceed if all Flow Control nodes with queues have space available for incoming tokens. Otherwise, the controller is stopped, but the speculative reads continue to execute and will (at some point in time) output and confirm the correct data, removing a token from the Flow Control node responsible for their Stage, and thus freeing up queue space. Please see [25] for further details.

Additional hardware (queues, token transition logic) will be inserted by Nymble into the statically scheduled controller only at the places required by the current application. This selective approach avoids the high overhead of relying on a general-purpose speculation support unit.

Now that we have discussed the PreCoRe microarchitecture and its automatic generation during hardware compilation, we can proceed to the last component of the solution, namely the multi-port memory system specialized to support speculative execution.

5 The MARC II Memory System

The multi-port cached memory system MARC II, initially presented in [17], has since been extended to support efficient operation of the PreCoRe mechanisms. PreCoRe relies on the memory subsystem to quickly satisfy the increased number of accesses due to execution replays. Note that MARC II deals strictly with non-speculative data, all value speculation occurs in PreCoRe itself. Furthermore, even though PreCoRe gives the appearance of single cycle memory reads (due to the value speculation), the scheme depends on low-latency replies from MARC II to quickly determine whether to commit a computation on confirmed values, or replay it due to a discovered misprediction.

5.1 Overview

The use of the spatially distributed computing paradigm on the adaptive computer also requires an appropriate parallel memory system. While some approaches rely purely on local on-chip memories (BlockRAMs), their limited size and lack of coherency protocols for shared accesses limits the scalability of the technique. Instead, we propose to use a shared memory system that gives the appearance of independent memory ports by providing each port with a distributed cache. Internal coherency mechanisms ensure a consistent view of all ports on the shared memory. Implementation-wise, we combine parallel on-chip BlockRAMs to realize fast caches, but still access the external off-chip main memory (shared with the software-programmable processor on the ACS) for bulk data.

The MARC line of memory systems has always aimed to provide multi-port operation supported by a dedicated cache infrastructure. In contrast, other ACS architectures often have at most a single port to external memory which is then explicitly allocated during scheduling to single memory operations. If they can actually serve multiple ports, they often have only very limited buffers (e.g., holding a DRAM row) as port-local storage. In contrast, MARC I [15] already gave multiple independent memory ports a coherent view of a shared multi-bank multi-port cache, allowing up to four parallel accesses. While the central shared cache avoided all coherency issues, it did not scale to larger numbers of ports and also limited the available clock frequency due to its fully-associative organization.

To lift both restrictions, MARC II (shown in Figure 12) instead relies on distributed per-port caches with a simpler, but faster direct mapped organization in on-chip BlockRAM. Since each MARC II per port-cache is larger than the MARC I central cache, the lower cache hit rates due to the direct mapped organization do not lead to slow downs. Since all of the caches operate independently, an large number of memory accesses can be served in parallel. Inter-port coherency is managed explicitly by a dedicated Coherency Bus (CB, described in the next section). As MARC I, MARC II is designed to isolate the hardware-independent core of the system from the device-dependent memory controllers (QDR2-SSRAM, DDR2/3-

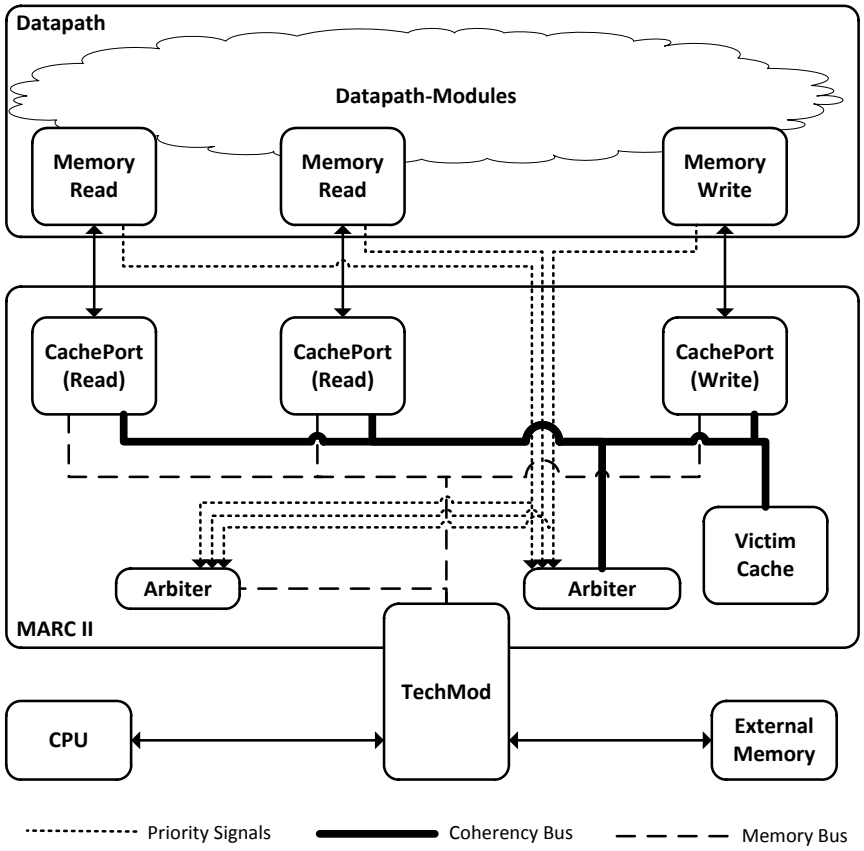

Fig. 12 Overview of the MARC II cache system

SDRAM, etc.), which are implemented as so-called TechMods. This allows the easy retargeting of MARC II-based accelerators to different ACS platforms.

5.2 Cache System and Coherency Protocol

Ensuring coherency between distributed caches is a difficult problem that has been the subject of much research, leading to protocols such as MSI, MESI, MOESI, etc. However, by tailoring the MARC II coherency mechanisms to the requirements of PreCoRe, we can employ a much simpler, low-overhead solution.

PreCoRe relies on load value speculation and does not support speculative writes. Thus, a single Write Port suffices in the memory system. All memory writes (being non-speculative) will have to be serialized through that port in program order (to

avoid violating WAW dependencies). This limitation is less severe than it appears, since conventional programs execute 3x . . . 6x as many loads as stores (measured in [11] for SPEC CPU 2006).

With the restriction to a single Write Port, we can employ a lightweight coherency protocol. Cache lines in a Read Port are either *valid* or *invalid*. In the Write Port, they are either *invalid* (the cache line is not present), *shared* (the cache line is present, and also present in at least one other Read Port cache) or *exclusive* (the cache line is present and no other cache has it). Note that the explicit *modified* state, common to general-purpose coherency protocols, is not required here, since the Write Port cache *only* holds modified lines.

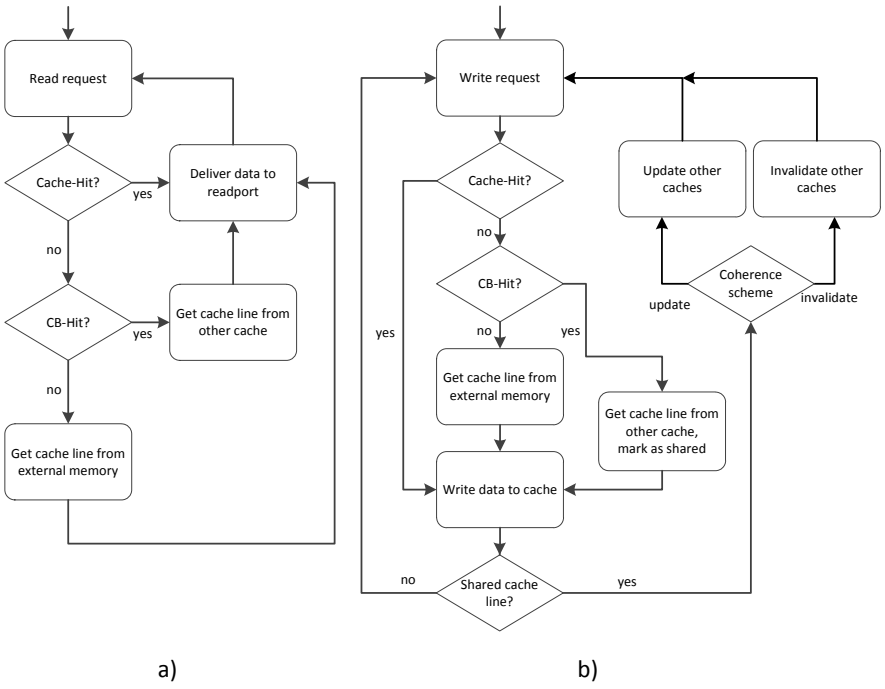

Fig. 13 Processing an access in a Read Port (a) and a Write Port (b)

Figure 13 sketches how requests from the datapath are handled by MARC II caches on memory reads (a) and memory writes (b).

Whenever a read request is executed, it is checked first whether it is a cache hit. If so, data can be provided in a single cycle from the Read Port cache without having to interact with any other shared resource. Thus, cache hits can be served completely independent of the actions of other ports. If the requested data is not available from its cache (local cache miss), the request is forwarded to all other caches connected to the CB by broadcast. Only if the request cannot be served by any of the other caches (remote cache miss), must the external memory be accessed.

The behavior for writes is slightly more complex. Again, the port first determines whether the necessary cache line is present. If so, the new data is inserted into the cache. If the modified line was shared with other Read Ports, coherency must be ensured. This can happen in one of two user-selectable modes: In invalidate mode, the Write Port tells the Read Ports holding an affected shared cache line to invalidate it. If the Read Ports later require the cache line again, it will be requested over the CB from the Write Port (which now holds the only copy). In update mode, the Write Port immediately transmits its modified cache line over the CB to all Read Ports holding the shared old versions (here, multiple copies of the line exist). If the write is a local cache miss, the Read Port caches are accessed via the CB. On a remote hit, the line is then marked as shared in the Write Port. Only if no port cache holds the data is the external memory accessed.

5.3 MARC II Support for PreCoRe Operations

Obviously, the paradigm of spatially distributing computation can only be maintained in the MARC II front-end. The rest of the infrastructure consists of time-multiplexed shared ressources (Coherency Bus, Memory Bus, TechMod, the actual external memory). The ports compete for access to these resources: In case of a local cache miss, the shared Coherency Bus must be accessed. On a remote cache miss, the request is forwarded to the shared external Memory Bus. If the external memory is in use (e.g., by the CPU), the access will have to wait until the memory becomes available for the RCU.

MARC II allows the accelerator to provide additional information on the priority of each access on a per-port basis: Each cache-port has its own priority-input, and an arbitration mechanism considers the given priorities of all pending requests when arbitrating the use of shared MARC II resources. This feature is used to apply the dynamic priority PreCoRe computes for each access (see Section 3.5) to influence the processing order of requests.

The displacement of cache lines in the distributed caches does not affect other caches, and thus is a less severe issue compared to cache line displacement in a single shared cache. However, the direct-mapped cache organization may cause frequent, undesirable cache displacements for some address sequences. In this case, the Memory Bus must be requested repeatedly to transfer the data from the external memory. Given the frequent memory accesses in PreCoRe, such displaced lines would lead to significantly longer replay times. By adding a small, fully-associative victim cache, these drawbacks can be reduced. The impact of a victim cache on performance and where it should be placed (L2 or L1) has been studied in detail for conventional processors [12]. In context of MARC II, the victim cache can be integrated seamlessly by attaching it to the Coherency Bus, where it just acts as another remote cache. This avoids the need for yet another communication network, and also keeps the access latency low by maintaining a single level of cache hierarchy.

MARC II also provides special support for the Selective Replay RAW-dependency resolution mechanism introduced in Section 3.4. Each Read Port has a (relatively small) Re-Execution CAM (RXC, see Figure 4.b) that holds the last *n* read addresses, where *n* is the PreCoRe speculation depth. The Write Port broadcasts the write addresses over the Coherency Bus (see Section 5.2) 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 signaled to the datapath in order to to initiate a replay.

6 Experimental Results

The ACS infrastructure proposed in this work has been implemented on the Xilinx ML507 development board using the Verilog hardware description language. Its core is a Virtex-5 FX FPGA, which is connected to various peripheral components, with a DDR2-SDRAM bank acting as external main memory. The reconfigurable fabric on the FPGA is used as RCU, the embedded PowerPC 440 as software-programmable processor. All benchmarks were compiled from C using the Nymble C-to-hardware compiler. The resulting RTL description was then synthesized using Synopsys Synplify Premier DP 9.6.2 and placed and routed with Xilinx ISE 11.1.

For evaluating the different components of the system, we used selected application benchmarks from well known benchmark suites (e.g., Mediabench [18], Honeywell ACS Suite [14]). The samples include the gf_multiply kernel from the Pegwit elliptic curve cryptography application, the quantization and wavelet transformation of the Versatility image compression application, and a luminance median filter. While the application benchmarks provide a good overview on the performance of the overall system, we also used synthetic hardware kernels to test specific features and characteristics of the system. The following paragraphs just summarize the actual results, please see [26, 25, 27] for the detailed measurements.

Each kernel was compiled twice, once with PreCoRe enabled, once with the original purely statically scheduled datapath. As previously discussed in Section 4.2, in the static version, a single cache miss stalls the entire datapath. Thus, all differences in performance are due to making better use of the hardware operators that are already present in the datapath.

Depending on the regularity of the input data, performance gains of up to 23% have been observed by employing load value speculation alone. Although successful speculation effectively hides the memory access latency of its particular access, even unsuccessful speculation may result in an improved execution time: The latency of later accesses may potentially be hidden by allowing them to execute earlier (instead of being stalled with the rest of the datapath). If the access executed early used a non-speculative read address, the data will be prefetched into the cache for use not only by the specific Read Port executing the early access, but also all other Ports that can retrieve it using the Coherency Bus (instead of accessing main memory again).

The dynamic priority computation discussed in Section 3.5 can lead to speed-ups 2%...25.5%. However, the best choice of weights for the computation is highly application dependent. Compiler support for selecting appropriate parameters automatically would be highly desirable here.

Despite being only a secondary effect of the actual read value speculation, the impact of prefetching should not be underestimated. As an experiment, we disabled the value-predictor, forcing it to always mispredict. Even in this crippled form, PreCoRe still executes reads as single cycle operations and avoids datapath-wide stalls, thus allowing prefetching to be performed. This prefetching-only version of PreCoRe yields a speed-up of 1.43x. Re-enabling the value-predictor reduces the execution time further to a total speed-up of 1.58x over the original statically scheduled version. For this specific benchmark, the prefetching made possible by the non-stalling mispredicting reads, and not the successful speculation, is actually responsible for most of the performance gain.

These benchmarks were constructed so that no RAW dependencies existed between accesses. If such dependencies cannot be ruled out (e.g., by using the C restrict keyword), the dynamic resolution mechanisms described in Section 3.4 need to be employed. For a synthetic benchmark that has a third of all speculative accesses violating RAW dependencies, the Selective resolution method (detecting overlapping addresses) requires up to 4% fewer clock cycles than the Universal resolution (that assumes all executed writes interfere with all reads). Adding a victim cache (Section 5.3) to speed-up replays further gains up to another 9% of clock cycles.

Combining the various features of PreCoRe, it was possible to achieve wall-clock improvements of up to 2.59x in our examples, *without* incurring any slow downs. This is a significant improvement over prior work such as [22] discussed in Section 1.

However, enabling PreCoRe has both an area as well as a clock frequency cost. The latter is not relevant for our experiments, since the maximum clock slowdown we observed (11% over the non-speculative versions) was either more than compensated by the PreCoRe speed-ups, or lead to a clock frequency that still exceeded the 100 MHz limit of the ML507 reference design. 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.

In contrast to the negligible clock slowdown, PreCoRe carries a significant area overhead (in our benchmarks: 1.45x...3.22x, counting slices). Much of this is due 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 mapable to FPGA shift-register primitives by the logic synthesis tool. Both of these issues could be addressed by adding the appropriate low-level optimization passes to Nymble.

7 Conclusion

We have presented a comprehensive approach to widening the memory bottleneck that is also starting to affect reconfigurable computing. It encompasses the microarchitectural mechanisms of the PreCoRe value speculation framework, the automatic generation of application-specific controllers implementing these techniques from C programs by the Nymble hardware compiler, and the run-time support for parallel memory accesses and quick execution replays provided by the MARC II memory system.

Our approach embraces the paradigm of spatially distributed computation, preferring to expend reconfigurable silicon area on application-specific computation support structures such as PreCoRe, instead of on general-purpose support mechanisms with diminishing efficiency, such as classical caches. With the ongoing trend towards ever larger reconfigurable devices, continued research in this area seems very promising.

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

References

[1] A. V. Aho, M. S. Lam et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2006.

[2] M. Budiu and S. C. Goldstein. Optimizing memory accesses for spatial computation. In *Proceedings of the Intl. Symp. on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pp. 216–227. IEEE Computer Society, 2003.

[3] M. Burtscher, B. G. Zorn et al. Hybrid Load-Value Predictors. *IEEE Trans. on Computers*, 51:759–774, 2002.

[4] T. J. Callahan, J. R. Hauser et al. The Garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, 2000.

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

[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. Springer-Verlag, London, UK, 1999.

[7] S. C. Group. Scale - a scalable compiler for analytical experiments, 2011.

[8] Z. Guo, W. Najjar et al. Efficient Hardware Code Generation for FPGAs. *ACM Trans. Archit. Code Optim. (TACO)*, 5(1):1–26, 2008.

[9] G.Weaver, B.Cahoon et al. Common Language Encoding Form (Clef) Design Document. Technical report, Department of Computer Science, University of Massachusetts, 1997.

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

[11] C. Isen, L. K. John et al. A Tale of Two Processors: Revisiting the RISC-CISC Debate. In *Proc. SPEC Benchmark Workshop*, pp. 57–76. 2009.

[12] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. of the 17th Annual Intl. Symp. on*

Computer Architecture, ISCA '90, pp. 364–373. ACM, New York, NY, USA, 1990.

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

[14] 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.

[15] H. Lange and A. Koch. An Execution Model for Hardware/Software Compilation and its System-Level Realization. In *Field Programmable Logic and Applications (FPL), 2007. Intl. Conf. on*, pp. 285 –292. 2007.

[16] H. Lange and A. Koch. Architectures and Execution Models for Hardware/Software Compilation and Their System-Level Realization. *Computers, IEEE Transactions on*, 59(10):1363 –1377, 2010.

[17] H. Lange, T. Wink et al. MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers. In *Design, Automation & Test in Europe (DATE), 2011 Conf. on*. 2011.

[18] C. Lee, M. Potkonjak et al. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proc., 30th Annual IEEE/ACM Intl. Symp. on*, pp. 330 –335. 1997.

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

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

[21] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1st edition, 1994.

[22] 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.

[23] A. Putnam, D. Bennett et al. CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures. In *Field Programmable Logic and Applications (FPL), 2008 Intl. Conf. on*, pp. 173 –178. 2008.

[24] 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.

[25] B. Thielmann, J. Huthmann et al. Evaluation of speculative execution techniques for high-level language to hardware compilation. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th Intl. Workshop on*, pp. 1 –8. 2011.

[26] B. Thielmann, J. Huthmann et al. Precore - A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation. In *Field Programmable Logic and Applications (FPL), 2011 Intl. Conf. on*, pp. 123 –129. 2011.

[27] B. Thielmann, T. Wink et al. RAP: More efficent eemory access in highly speculative execution on reconfigurable adaptive computers. In *ReConFigurable Computing and FPGAs (ReConFig), 2011 Intl. Conf. on*. 2011.

[28] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Microarchitecture, 1997. Proceedings., 30th Annual IEEE/ACM Intl. Symp. on*, pp. 281 –290. 1997.

[29] 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.