# Memory Latency Hiding by Load Value Speculation for Reconfigurable Computers

BENJAMIN THIELMANN, Technische Universität Braunschweig
JENS HUTHMANN and ANDREAS KOCH, Technische Universität Darmstadt

Load value speculation has long been proposed as a method to hide the latency of memory accesses. It has seen very limited use in actual processors, often due to the high overhead of reexecuting misspeculated computations. We present PreCoRe, a framework capable of generating application-specific microarchitectures supporting load value speculation on reconfigurable computers. The article examines the lightweight speculation and replay mechanisms, the architecture of the actual data value prediction units as well as the impact on the nonspeculative parts of the memory system. In experiments, using PreCoRe has achieved speedups of up to 2.48 times over nonspeculative implementations.

Categories and Subject Descriptors: B.5.1 [**Register-Transfer-Level Implementation**]: Design—*Control design, Data-path design*; B.5.1 [**Register-Transfer-Level Implementation**]: Design Aids—*Automatic synthesis*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures, Heterogeneous systems*

General Terms: Design, Performance

Additional Key Words and Phrases: FPGA, high-level language compilation, speculative execution, memory interface

**ACM Reference Format:**
Thielmann, B., Huthmann, J., Koch, A. 2012. Memory Latency Hiding by Load Value Speculation for Reconfigurable Computers. ACM Trans. Reconfig. Technol. Syst. 5, 3, Article 13 (October 2012), 14 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

## 1. INTRODUCTION

Adaptive computing systems (ACS) combine software-programmable processors (SPPs) and reconfigurable compute units (RCU) to efficiently provide computing performance. The time-critical parts of a computation, so-called *kernels*, are realized as hardware accelerators on the RCU, while the SPP handles the noncritical application and system management tasks. Considerable research effort has been invested in making ACSs more accessible, specifically on automatic compilers [Callahan 2002; Li et al. 2000; Kasprzyk and Koch 2005] aiming to translate high-level language description into RCU implementations. With the improving quality of these compile flows, classical computer architecture bottlenecks such as memory latency also become more pronounced in an ACS architecture. According to Kaeli and Yew [2005], 20% of a typical program's instructions are memory accesses, but they require up to 100x of the execution time of the nonmemory (scalar) operations.

Among the solutions that have already been proposed are increasing the memory bandwidth by parallel localized memories [Gädke-Lütjens et al. 2010], or by using coherent distributed caches in a shared memory system [Lange et al. 2011].

As an orthogonal alternative to these techniques, we propose the use of *load value speculation* to hide the latency of memory reads. Reads thus become fixed-latency operations that always supply a result within a single clock cycle.This capability requires special support in the RCU microarchitecture, since misspeculated values must be recognized at some point in time and the affected earlier computations must be reexecuted (replayed) with the correct input values. When performing a replay, it is desirable to limit the extent of the reexecuted operations. Ideally, only the specific operations "poisoned" by the misspeculated values need to be replayed. However, since this is equivalent to a dynamically scheduled execution model, implementing such a fine

replay granularity requires complex hardware [Gädke-Lütjens 2011; Gädke-Lütjens et al. 2010].

Our PreCoRe framework *Pre*dicts, *Co*mmits, and *Re*plays at the granularity of datapath *stages*, similar to the clock cycles used in a statically scheduled microarchitecture, and aims to never slow down the execution of the datapath over the nonspeculative version.

This article first gives an overview over three of PreCoRe's main concepts: The replay (reexecution) control mechanisms (Section 3.1), the associated speculation queues (that resupply the appropriate data items, Section 3.2), and the actual data value predictors themselves (Section 3.3). These topics have been individually presented in greater detail in Thielmann et al. [2011b] and Thielmann et al. [2011a], but will be discussed here in a common context. Beyond this prior work, we show new results on the energy-efficiency of the PreCoRe architecture and examine how PreCoRe complements compile-time scalarization techniques for memory access optimization. Furthermore, we introduce a number of techniques to reduce the cost of replays on misspeculated values (Section 5 and 7.3) and consider their impact on the interaction between the memory system and the speculation mechanisms. All of the techniques presented have been implemented in our hardware/software cocompiler Nymble as extensions to simple statically-scheduled datapaths, and all of the benchmarks have been mapped to and executed on a current ACS hardware platform.

## 2. RELATED WORK

Even assuming support for unlimited instruction-level parallelism (ILP) in reconfigurable hardware, memory read data dependencies severely limit the degree of ILP achievable in practice to between tens to (at most) hundreds of instructions [Hennessy and Patterson 2003].

Lipasti et al. [1996] proposed load value speculation to resolve these read data dependencies speculatively, allowing computations to continue using the speculated values without waiting for the memory system to return the actual (possibly cached) memory contents.

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 [González and González 1998; 1999].

Due to the associated hardware overhead for general-purpose solutions, value speculation has only been used to a limited extent in actual processors. Mock et al. [2005] modified a compiler to force value speculation where possible on the Intel Itanium 2 CPU architecture. Their scheme relied on hardware support in the form of the Itanium's Advanced Load Address Table (ALAT) [McNairy and Soltis 2003]. However, the ALAT does not operate autonomously, but has to be explicitly controlled by software code. At best, they achieved speedups of 10% for read value speculation using this approach. However, they also observed slowdowns of up to 5% under adverse conditions.

Given the current concern of computer architects with energy efficiency, the power impact of speculation has to be carefully considered. Sam and Burtscher [2005] introduced a metric to compare the energy-efficiency of speculative architectures. They state that the added complexity of an intricate prediction scheme often increases the energy consumption of the processor at a higher rate than it increases performance. In such architectures, when energy consumption is taken into consideration, complex

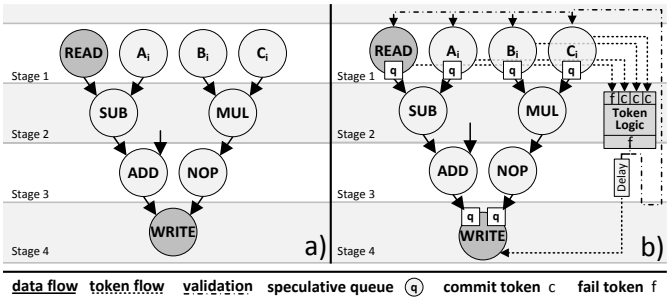


Fig. 1. Datapath and speculation token processing.

predictors do not provide a good energy-performance trade-off. We will consider the energy characteristics of our approach in Section 6.

## 3. SPECULATION FRAMEWORK

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

On top of these foundations, different kinds of speculation can be realized, for instance, the system supports control, data value, and data dependency speculation. For purposes of this work, we will concentrate on the use of load value speculation to hide memory access latencies.

### 3.1. Token-Based Speculation Control

PreCoRe extends a simple statically scheduled datapath by creating application-specific speculation control logic. Using this logic, the execution of variable latency operators such as cached reads and writes, no longer halts the datapath on violated latency expectations (here: due to a cache miss instead of the expected single-cycle cache hit), but allows execution to proceed using speculated values. This gives the appearance of read and write being fixed-latency operators that always produce/accept data after a single cycle.

However, the propagation of these speculated values in the datapath must be limited to always allow reversal of incorrect operations and their associated sideeffects. For this reason, the irreversible write operations form a *speculation boundary* in the current approach. Here, the speculated data *must* have been confirmed as correct for the write to proceed. Otherwise, the execution leading up to the write has to be replayed with the correct data, which now has been delivered from the memory system.

*Successful Value Speculation*. Consider the example in Figure 1(a). Before executing the WRITE, the system has to ensure that the data to be written is confirmed as being correct. This is done by its originating node, here the READ node, which is the sole source of speculated data in the current PreCoRe prototype. For the confirmation, the READ node internally retains its speculatively output values, while the WRITE node buffers incoming values dependent on the speculated read value in an input queue. If the oldest speculative value retained in the READ node matches the value actually returned from memory, the WRITE nodes holding data dependent on that value in their input queues are allowed to proceed. Due to their input queues, the WRITE nodes also give the appearance of being single-cycle operations.

PreCoRe uses a token-based mechanism to actually implement this behavior. The underlying Token Logic is generated in an application-specific manner and can thus be more efficient than a general-purpose speculation management unit. An example is shown in Figure 1(b). Additional edges show the flow of tokens and validation signals. The latter indicate that all of the operators in a stage (initially equivalent to a statically scheduled clock cycle) have output correct values and no longer need to retain old data for eventual replays in speculation queues (described in greater detail in Section 3.2).

When the speculated and actual data values have been determined to match, the READ node indicates the successful speculation by sending out a Commit-Token (shown as C in the Figure) to the Token Logic. In contrast to operator-level speculation, the token is not directly forwarded to the WRITE node. Instead, the speculation status is computed per stage: The speculation status signals of all operators in a stage (even the nonspeculative ones, since they might have operated on speculated values) are combined: Only if all operators in a stage indicate the presence of correctly speculated data, is a corresponding C-Token forwarded to the operator at the speculation boundary (the WRITE). There, it confirms the correctness of the last datum with undetermined speculation status. Note that speculated values and their corresponding tokens are associated only by their sequential order. While this prevents out-of-order processing of memory requests, it also avoids the need for additional administrative information such as Transaction IDs.

However, a token is allowed to temporarily overtake its corresponding datum up to the next synchronization point (see Section 3.2), since the speculation status of an entire stage is only affected if it actually contains a speculative operator (in our approach, a READ node). Otherwise, the speculation status will just depend on the stage inputs (nonspeculative, if no speculative values were input; speculative, if even a single input to the stage was speculative). Since the stages 2 and 3 do not contain READs, an incoming token can be directly forwarded to the WRITE in stage 4, where it will wait in a token queue for the correctly speculated value to arrive and allow the WRITE to proceed. Fast token forwarding is crucial for benefitting from successful speculation. Note that, simultaneously with this process, pipelining might have lead to the generation of more speculative values in the READ, which continue to flow into the succeeding stages.

*Failed Value Speculation*. In the case of a failed speculation, two actions have to be taken: All data values depending on the misspeculated value have to be deleted (their lack also prevents the WRITE from executing), and the affected computations have to be restarted (replayed), this time with the correct nonspeculative result of the READ.

To this end, the token logic first determines that the misspeculated READ in Figure 1(b) leads to the entire stage 1 failing in its speculation. In contrast to the C-Token discussed before, which had an immediate effect, the Fail token (F-Token) is delayed by the number of stages between the source of the speculated value and the WRITE marking the speculation boundary (in this case, two stages, leading to a delay of two clock cycles). To be more precise, the token is delayed by two clock cycles when the datapath is actually computing. If it were stalled (e.g., all speculative values have reached speculation boundaries, but could not be confirmed yet by memory accesses because the memory system was busy), these stall cycles would not count towards the required F-Token delay cycles. This ensures that all intermediate values computed based on the misspeculated value in stages 2 and 3 have now ended up in the input queues of the WRITE operation in stage 4, and will be held there since no corresponding C- or F-Token for them has been received earlier. The appropriately delayed F-Token then arrives to resolve this situation, deleting the two sets of incorrectly computed intermediate results from the operator input queues and prevents the WRITE from executing.

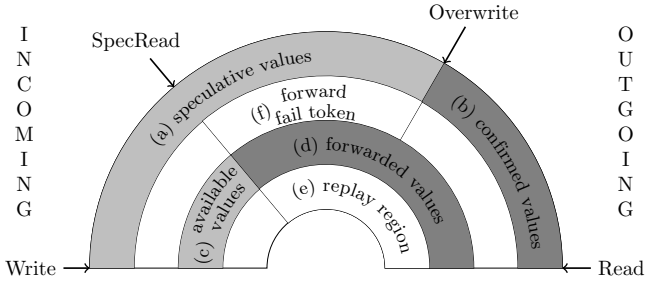


Fig. 2. Attributed regions in speculative queues.

The replay starts immediately after the misspeculation has been discovered. READ outputs the correct data, all other nodes in stage 1 reoutput their last set of results (which may still be speculative themselves!) from their output queues, and perform the computations in stages 2 and 3 again.

### 3.2. Speculation Queue Management

As described in the prior section, speculation queues at operator outputs play a significant role in the speculation mechanism. They are responsible for providing succeeding stages with data (speculated or accurate), holding data until the need for a later replay has been ruled out, and to discard misspeculated values. Furthermore, they synchronize the asynchronously flowing value and token sequences, as tokens cannot overtake values through an output queue. The internal operation of a speculation queue is thus somewhat more complicated than that of simple conventional queues, as shown in Figure 2.

The speculative output queue consists of two circular buffers (one for values, one for tokens) organized as two queues:

In the value queue, each entry between the Read and Write pointers has various attributes, which determine how the data is processed. At the incoming side, values are either speculative (a), or were confirmed by incoming tokens (b). Since all data values are committed sequentially, and no additional committed data may arrive once the first speculative data entered the queue, these regions are contiguous. At the outgoing side, values have either been passed on to a succeeding operator (d), or are awaiting transmission (c). Again, these regions are contiguous.

Two additional pointers are required to implement the extra regions: The region starting with and extending beyond the OverwriteIndex are those positions holding speculated values (a), they will need to be overwritten with new values, if they were misspeculated (these may again be speculative values). To let speculative values flow to succeeding operators, the SpecRead pointer is used. All values below that pointer (d) have been passed to succeeding operators, but they might be misspeculated values. Thus, they are still included in the replay region (e). The basic queue behavior is realized using the Write pointer to insert newly incoming speculative data at the head of region (a), and the Read pointer to remove a confirmed and forwarded value at the tail of region (d) after the entire stage has been confirmed.

Values and tokens are synchronized strictly by their sequential order, not by being in the same position of the queues. The token queue must have twice the length of the value queue to handle the corner case of an incoming sequence of alternating C- and F-Tokens. Incoming fail tokens are inserted/removed as in a conventional queue, but also need to be forwarded to the subsequent stage if a misspeculated value has previously been provided to the subsequent stage (f).

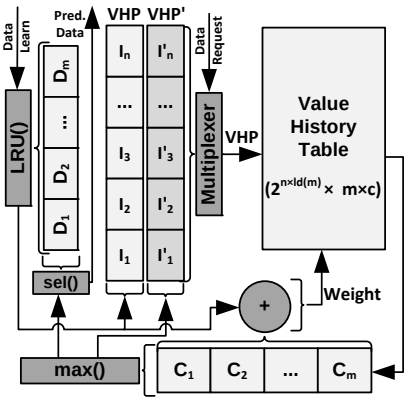


Fig. 3. Local history-based load value predictor

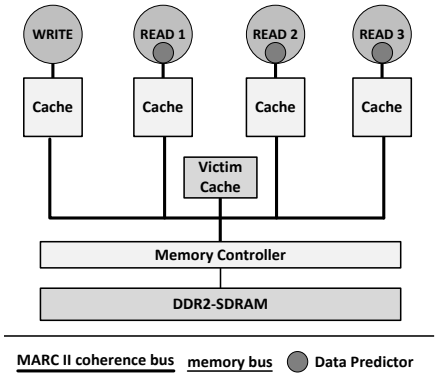


Fig. 4. MARC II memory system extended with read value speculation and victim cache

Input queues behave similarly to output queues, with one exception: they do not require validation of committed data. Instead, values are removed from the queues as soon as their C-Tokens arrive. A complete example for both token handling and queue management is given in Thielmann et al. [2011a].

### 3.3. Load Value Predictor Architecture

Figure 3 shows the basic architecture of our value predictor. It follows a two-level finite-context scheme building on concepts initially used in branch prediction. Value predictions are based on exploiting a correlation of a stored history of prior data values and future values [Wang and Franklin 1997]. The actual nature of the correlation is flexible and highly parametrized: We use the same basic architecture to realize both last value prediction (which predicts a future value by selecting it from a set of previously observed values, for instance, *23-7-42-23-7-42*) and stride prediction (which extrapolates a new value from a sequence of previously known strides, for instance, from the strides *4-4-8-4*, the sequence *0-4-8-16-20-24-28-36-40* is predicted). The complete PreCoRe value prediction unit consists of separate last-value and stride-predictors, operated in tournament mode. For brevity, we discuss only value speculation here, last-value prediction is performed analogously.

To improve prediction accuracy, the predictor is *trained speculatively* with the data it provided as being likely, but that was not yet confirmed correct. Thus, the prediction strategy can only be switched to the other predictor once an actual misprediction has been discovered. Note that the speculator itself is only parametrized at compile time, it is not dynamically reconfigured.

For the pattern database, the predictor not only keeps track of the last $m$ different values $D_1, \ldots, D_m$ in a least-recently-used fashion, but also of the $n$-element sequence $I_1, \ldots, I_n$ in which these values occurred (the Value History Pattern, VHP). An element of $I$ is an index reference to an actual value in $D$. The entire sequence $I$, which consists of $n$ subfields of $\log_2 m$ bits each, is used to index the Value History Table (VHT) to determine which of the known values is the most likely result of the prediction. An entry of the VHT expresses the likelihood for each of the known values $D_i$ as a $c$-bit unsigned value $C_i$, with the highest value indicating the most likely value (on ties, the smallest $i$ wins). Thus, the VHT is accessed by a $n \log_2 m$-bit wide address and stores words of width $mc$ bits. On reset, each VHT counter is initialized to the value $2^{c-1}$, indicating a probability of $\approx 50\%$. The stored state for VHT and VHP (and thus the time required for learning to make accurate predictions) grows linearly with $n$ and logarithmically

with $m$. In a future refinement, the predictor could be configured automatically for each application, with the compiler selecting $n$ and $m$.

The actual prediction process has to take mispredictions into account. Thus, we keep two copies of the VHP: The master VHP $I$ (shown in blue in Figure 3) holds *only* values that were confirmed as being correct by the memory system, but may be outdated with respect to the actual execution (since it might take a while for the memory system to confirm/refute the correctness of a value). The shadow VHP' $I'$ (shown in red in the Figure) also includes speculated values of *unknown* correctness, but accurately reflects the current progress of the execution. All predictions are based on the shadow VHP' until a misprediction occurs. In the datapath, this would lead to a replay using the last values not already proven incorrect. In the predictor, the same effect is achieved by copying the master VHP (holding correct values) to the shadow VHP' (to base the next predictions on the new values). Please see Thielmann et al. [2011a] for detailed descriptions and a complete example of the predictor operation.

## 4. MEMORY SYSTEM INTERACTION

Load value-prediction relies on a high-performance memory system to confirm/refute the speculated values as quickly as possible. We employ MARC II [Lange et al. 2011], a highly parametrized framework supplying parallel memory ports to the datapath. Each port can be individually configured for streaming (regular) and caching (irregular) operation. The more general case of irregular accesses is supported by distributed per-port caches, which are kept coherent using configurable application-specific synchronization mechanisms.

As shown in Figure 4, for use with PreCoRe, the actual data prediction units are inserted between the datapath-side read ports and the MARC II-internal per-port caches, leaving the rest of the MARC II architecture untouched. With this modification, read requests issued by the datapath on the RCU are simultaneously passed both to the per-port cache and to the data prediction unit. Thus, the datapath will always receive a reply within just a single cycle of latency: Either the actual data will have been present in the cache (cache hit) and can be returned, or the data prediction unit supplies a speculated value for the read request on a cache miss.

Among the many configurable MARC II parameters is the presence and organization of a Victim Cache. We will examine its interactions with the speculation cost in the next section.

## 5. REDUCING THE COST OF MISPREDICTIONS

The minimization of the penalty of mispredictions is crucial for the overall performance of a system relying on speculative execution. A key aspect of reducing this penalty is attempting to serve replayed memory accesses from cache and avoiding main memory accesses.

This can be achieved in a number of different ways: An obvious approach are larger caches. However, this will eventually lead to scaling problems due to increased area and delay (the latter especially for fully associative caches). The multiport caches in MARC II would suffer even more from such scaling, as all size increases would be multiplied by the number of ports.

Looking at the replay cost problem more closely, we want to ensure that data that has already been fetched from main memory to confirm/refute the initial speculative access, remains present for a later replay. A classical way to achieve this in a scalable fashion is a victim cache (VC, Jouppi [1990]). This is a relatively small, but fully associative cache that buffers blocks displaced from upstream caches. One of the configurable features of MARC II is the presence and size of such a VC per cluster, shared among the per-port distributed coherent L1 caches. Enabling the VC has minimal area overhead, since it

Table I. Hardware Area and Maximum Clock Frequency without/with Data Speculation (n.spec/spec)

| Kernel | FPGA Area | | | | Max. Clock Freq. | | Comparison | |
|---|---|---|---|---|---|---|---|---|
| | #LUTs | | #Registers | | (MHz) | | Slices | Power |
| | n. spec | spec | n. spec | spec | n. spec | spec | overhead | overhead |
| array_add | 10141 | 14717 | 1246 | 2948 | 106.90 | 96.30 | 1.45x | 1.20x |
| array_sum | 11159 | 22363 | 1579 | 6402 | 105.10 | 94.60 | 2.00x | 1.12x |
| bintree_search | 11129 | 19782 | 1570 | 5795 | 105.70 | 100.10 | 1.78x | 1.30x |
| gf_multiply | 11918 | 22790 | 1702 | 6459 | 102.80 | 101.30 | 1.91x | 1.29x |
| median_filter_row | 12895 | 28333 | 2458 | 9909 | 106.40 | 102.20 | 2.20x | 1.32x |
| median_filter_col | 12998 | 28391 | 2529 | 10325 | 106.00 | 100.30 | 2.19x | 1.33x |
| pointer_chase | 11979 | 20637 | 1478 | 10208 | 106.40 | 98.90 | 1.72x | 1.27x |
| simple_read | 10241 | 14703 | 1484 | 3639 | 105.80 | 103.20 | 1.45x | 1.31x |
| versatility_quant. | 12351 | 39716 | 5390 | 18495 | 105.70 | 97.40 | 3.22x | 1.30x |
| versatility_fcdf22 | 12055 | 32284 | 1889 | 9863 | 105.20 | 93.80 | 2.68x | 1.31x |

Table II. Runtime without/with Data Speculation (n.spec/spec)

| Kernel | Runtime | | | | Comparison | |
|---|---|---|---|---|---|---|
| | #Cycles | | $\mu$s at max. freq. | | Speedup | Energy |
| | n. spec | spec | n. spec | spec | | efficiency |
| array_add | 6194 | 3923 | 57.94 | 40.74 | 1.42x | 1.18x |
| array_sum | 1786 | 946 | 16.99 | 10.00 | 1.69x | 1.35x |
| bintree_search | 3497 | 3359 | 33.08 | 33.56 | 0.99x | 0.76x |
| gf_multiply | 2510 | 2482 | 24.50 | 24.42 | 1.00x | 0.78x |
| median_filter_row | 296409 | 114650 | 2785.80 | 1121.82 | 2.48x | 1.88x |
| median_filter_col | 1054736 | 666554 | 9950.34 | 6665.24 | 1.50x | 1.13x |
| pointer_chase | 4087 | 3650 | 38.41 | 36.91 | 1.04x | 0.82x |
| simple_read | 19615 | 14150 | 185.40 | 135.41 | 1.37x | 1.05x |
| versatility_quant. | 96771 | 50746 | 915.53 | 521.01 | 1.76x | 1.35x |
| versatility_fcdf22 | 43633 | 20573 | 414.76 | 219.33 | 1.89x | 1.44x |

attaches to MARC II's existing intra-cluster coherency network. By sizing the VC to the current speculation depth (how many unconfirmed values can remain in flight, typically 8 or 16), all potentially displaced cache blocks can be caught and will be available very quickly for the replays.

Interestingly, in certain cases, another even lighter-weight approach suffices to ensure quick replays. It relies on the static (compile-time) memory operation prioritization supported by both Nymble and MARC II. While the individual memory ports supplied by a MARC II instance can all be accessed in parallel, they ports share some buses and access to the single main memory itself. These resources are time-multiplexed between ports, using the MARC II port number as the static priority when arbitrating parallel accesses (lower port numbers have higher priority). As will be shown in Section 7.3), for datapaths without loop-carried dependencies an appropriate assignment of memory operators to ports can achieve fast replays without the need for a VC. If loop-carried dependencies are present, though, even a small VC can be very beneficial.

## 6. EXPERIMENTAL RESULTS

In this section we show the raw characteristics (including performance, area, and power) of a number of benchmark applications. In all cases we compare implementations using PreCoRe to purely statically scheduled versions.

### 6.1. Performance

Tables I and II show the results of executing a number of benchmark programs using our PreCoRe-enhanced Nymble compiler. The RTL Verilog created by Nymble was then synthesized for a Xilinx Virtex-5 FX device using Synopsys Synplify Premier DP 9.6.2 and Xilinx ISE 11.1. Our target ACS is based on the Xilinx ML507 development board enhanced with the FastPath, FastLane, and AISLE [Lange and Koch 2010] features.

As a baseline for our comparison (both for performance and area), we used MARC II to provide cached accesses to the FPGA-external DDR2-SDRAM, with the memory ports being organized as a single coherency cluster. In the nonspeculative case (using only static scheduling), we halt the entire datapath if a memory access cannot be resolved within a single cycle.

array_add increments each element of an array, without loop-carried dependencies. array_sum sums up all elements of an array, with loop-carried dependencies. bintree_search searches a binary tree. gf_multiply is part of the Pegwit elliptic curve cryptography application in MediaBench [Lee et al. 1997]. median_filter_row and _col realize a luminance median filter. Blocks of 9 pixels are read row/column-wise and the median of luminance is written to the center pixel. The column-based processing shows the effectiveness of data speculation when cache efficiency decreases due to unsuitable access patterns. pointer_chase processes a randomly linked list, writing to every second element. simple_read sums all values of an array, and thus has a loop-carried dependency. versatility_quantization and versatility_fcdf22 are the quantization and Wavelet steps of an image compression benchmark [Kumar et al. 2000].

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

Despite the area increase of up to 3.2x, we observed an increase in power consumption of only 1.12x. . . 1.32x for the speculative over the nonspeculative versions. This is due to most of the power being drawn by the multiport MARC II memory system, with only a small fraction of the power actually consumed in the speculative datapaths.

When looking at the energy required, the performance gains (shorter runtimes) due to successful speculation can actually lead to an improved energy efficiency over the nonspeculative versions (by up to 1.44x for versatility_fcdf22) despite the higher power draw. On the other hand, applications with more misspeculations (not achieving shorter runtimes) pay a price in energy efficiency, which has been observed to drop down to 0.76 for bintree_search compared to the nonspeculative version.

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

## 7. RESULT ANALYSIS

Here, we interpret the raw performance data presented in Section 6. We discuss the impact of both first and second order effects and compare PreCoRe with a compile-time optimization technique for entirely avoiding memory accesses.

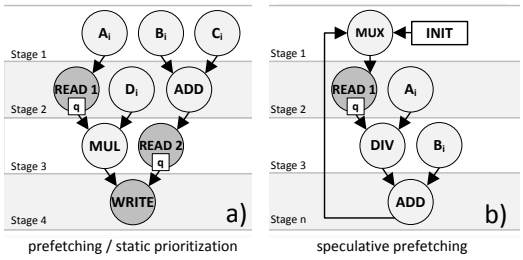


Fig. 5. Interaction of speculation with prefetching.

Table III. Execution Time vs. Predictor Accuracy

| Runtime | Hit Rate | | | Speed-up |
|---|---|---|---|---|
| #Cycles | Total #Commits | Total #Fails | % Acc. | % |
| 33612 | 991 | 12 | 99 | 23 |
| 33733 | 969 | 34 | 97 | 22 |
| 34025 | 933 | 70 | 93 | 21 |
| 34331 | 877 | 126 | 87 | 20 |
| 35127 | 790 | 213 | 79 | 17 |
| 35550 | 734 | 269 | 73 | 16 |
| 35880 | 692 | 311 | 69 | 15 |
| 36361 | 628 | 375 | 63 | 13 |
| 36841 | 541 | 462 | 54 | 12 |
| 38138 | 398 | 605 | 40 | 8 |
| 39447 | 182 | 821 | 18 | 4 |
| 41201 | 0 | 1003 | 0 | 0 |

### 7.1. Effects of Prefetching

In the original, purely statically scheduled RCU shown in Section 3, a single cache miss would stall the entire datapath, halting even operations not actually dependent on the result of the cache-missing read. With PreCoRe, the datapath is *not* stalled, since a read always returns data within a single clock cycle. This enables memory prefetching, even if the speculated read value turns out to be wrong.

Figure 5(a) shows an example for this situation: Assume that READ1 suffers a cache miss and returns speculative data. By not stalling the datapath, READ2 is allowed to proceed, prefetching data from a nonspeculated address. Even if READ1 speculated incorrectly and a replay would be required, READ2 will have prefetched the correct line into its own cache by then (assuming it had a cache miss at all). This would not have been possible in the static datapath, since READ2 would only be started after READ1 had completed processing its own cache miss.

The scope of prefetching can be widened even further when considering prefetching from a *speculative address*, which is also supported in PreCoRe. Figure 5(b) shows an indirect READ, with the address being a value-speculated, loop-carried dependency from a prior loop iteration. Prefetching can be performed here, too, if the address sequence has a form predictable by the stride predictor of Section 3.3. In practice, this could occur, for instance, if an array of pointers into an array of structures is being processed.

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 predict incorrect values, in median_filter_col. Even in this crippled form, PreCoRe still executes reads as single cycle operations and avoids datapath-wide stalls (the scenario shown in Figure 5(a)). Prefetching can still be performed under these conditions. The static version of the kernel requires 1,054,736 cycles to execute, while the prefetching-only version using PreCoRe is accelerated to 738,607 cycles, yielding a speedup of 1.43x. Additionally enabling the value-predictor reduces the execution time further to 666,554 cycles, a total speedup of 1.58x over the original static version. For this example, the prefetching enabled by the fixed-latency speculated reads, not the speculated values themselves, is actually responsible for most of the performance gain.

### 7.2. Effects of Successful Load Value Speculation

While the previous section concentrated on using PreCoRe to enable prefetching, this section discusses the effects of the accuracy of the value-prediction on execution. To this end, we parametrize pointer_chase to generate a range of predictor accuracies. The

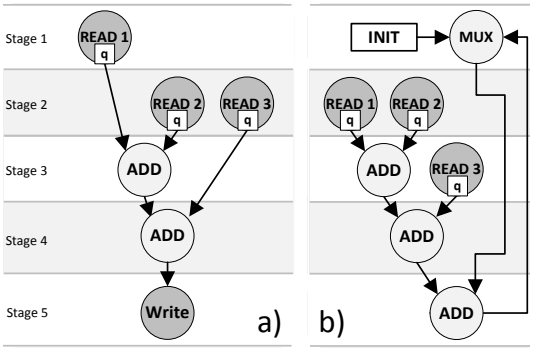


Fig. 6. Scenarios for prefetching interfering with rapid replays.

Table IV. Reducing misprediction costs by static port priority and victim cache

| victim cache entries | Static Priority Mapping [Normalized Clock Cycles] Fig 5.a In-Order | Fig 5.a Reverse | Fig 5.b In-Order | Fig 5.b Reverse |
|---|---|---|---|---|
| 0 | 0.57 | 1.00 | 1.00 | 0.40 |
| 4 | 0.57 | 1.00 | 1.00 | 0.40 |
| 8 | 0.57 | 1.00 | 0.98 | 0.40 |
| 16 | 0.57 | 1.00 | 0.93 | 0.40 |
| 24 | 0.57 | 1.00 | 0.83 | 0.32 |
| 32 | 0.57 | 0.57 | 0.32 | 0.32 |
| non-spec. | 0.59 | 0.59 | 0.33 | 0.33 |

application has a basic structure similar to that shown for Figure 5(b), with the address in the current iteration being a loop-carried data dependency of the prior iteration (that list element's `next` pointer).

Without value speculation, a new iteration can only be started once a prior iteration has completed retrieving the `next` pointer. Using PreCoRe, the READ in the current iteration can be started immediately using the speculative value returned from the prior iteration as address for a prefetch.

Table III shows the impact of predictor accuracy for such a scenario. The extreme cases (completely accurate and inaccurate predictions) are shown in the first and last rows, respectively. Depending on the regularity of the input data (in this case, the regularity of the linked elements in memory), performance gains of up to 23% are possible. The effect is magnified if an even longer backwards edge implies a longer initiation interval (II) of the loop: The successful use of speculated addresses for indirect accesses can reduce the II, and increase throughput correspondingly.

### 7.3. Misprediction Costs and Prefetching

Despite the advantages of prefetching, it can interfere with misprediction replays by competing for cache entries. In the worst case, the prefetched lines have replaced those that would be required for a rapid replay, drastically increasing misprediction costs. As suggested in Section 5, this situation can be avoided by making use of static port priority assignments and the presence of a small victim cache (VC). For the following experiment, we have set up the prefetch address sequences to induce conflict misses within each read operator's direct-mapped MARC II cache. Furthermore, we force the speculation to fail continuously, stressing the replay mechanism. This allows us to evaluate the different schemes for reducing replay costs.

Figure 6(a) shows a simple datapath without loop-carried dependencies. A misspeculation of READ1 will also lead to the replay of READ2 and READ3 (due to the replays occurring at Stage-granularity). When assigning the read operators in program order to the cache ports (earlier reads get a higher priority), even the continuously failing RCU executes still faster than the nonspeculative version (Table IV, 0.57 vs. 0.59 normalized clock cycles). Replay costs are kept low here purely by the choice of port priorities, adding a VC does not increase performance. As a counterexample, assigning the operators in a Reverse (bottom-up) order to the ports would lead to a significant slowdown (almost by 1.75x), which could only be corrected by enabling a VC large enough to hold all of the displaced lines for fast replays. The size of the required VC is due to READ2 and READ3 both needing to be replayed once READ1 misspeculated. Thus, for the speculation

```
for(i=0;i<n;i++) {
 sum += a[i] + a[i+1];
}
```

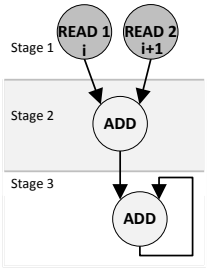


Fig. 7. `array_sum` before scalarization.

```
a0 = a[0];
for(i=0;i<n;i++) {
 a1 = a[i+1];
 sum += a0 + a1;
 a0 = a1;
}
```

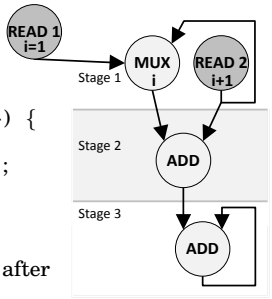


Fig. 8. `array_sum` after scalarization.

depth of 16 used here, a VC of 32 entries would be required to enable rapid replays for a poor choice of port priorities.

The situation becomes more complex when considering RCUs with loop-carried dependencies (Figure 6(b)). Here, a misspeculated READ3 in iteration $i$ will lead to the replay of READ1 and READ2 already started for iteration $i + 1$. Even though the latter are not directly data-dependent on READ3, the misspeculated value of READ3 propagates to the computation in Stage 1, which will then replay the succeeding Stage 2. With such an inter-iteration dependency, the Reverse operator to port priority assignment actually leads to a faster execution time over the In-Order scheme (by 2.5x). However, even when using the more suitable scheme, the replays have become so expensive that the speculative version is 1.2x slower than the nonspeculative one. Resolving this discrepancy does require the use of a VC: With the Reverse priority assignment, the remaining displaced lines can be caught with 24 VC entries, enabling a small speedup of the continuously failing speculative version over the nonspeculative one.

### 7.4. Comparison with Scalarization

Scalarization is a compile-time optimization technique which transforms array operations to avoid unnecessary memory traffic by buffering and reusing fetched data [Callahan et al. 1990] in registers. However, due to the limited number of registers of conventional CPUs, aggressive application of the scalarization technique may also reduce performance due to increased pressure on the register allocator. With the abundance of registers on modern RCUs, they form a much more promising target for aggressive scalarization [Budiu and Goldstein 2005].

While both PreCoRe and scalarization attempt to widen the memory bottleneck, they are orthogonal in their use: Scalarization is able to completely avoid some accesses under specific circumstances (suitable static code structure), but incurs the full memory access latency for the remaining accesses in case of a cache miss. PreCoRe always accesses memory, but can potentially hide the access latency by exploiting dynamic data value correlations and prefetching (Section 7.1).

For a comparison of the techniques, consider the program array_sum described in Figure 7. Using scalarization, the memory access *a[i]* inside the loop can be removed, the value of the memory access *a[i+1]* can be reused from the previous iteration by buffering it in a local register. Only for the first iteration, an initial memory access for pre-loading *a[0]* into the register *a0*, needs to be performed outside the loop. The reordered code and data path, resulting from scalarization, is shown in Figure 8.

A simulation of a scalarized version of array_sum shows a speedup of 1.25x. The execution time is reduced by the time previously required for the retrieval of data, which is now kept in a local register. Using PreCoRe, the achievable speedup is highly dependent on the predictability of the load values and lies between 1.15x... 1.47x. The lower bound corresponds to a totally unpredictable sequence of values (but still allowing

prefetching of *a[i+1]*), the upper one corresponds to an accurately predictable sequence. In contrast to scalarization, PreCoRe does not rely on a specific code structure and is thus also applicable to speedup benchmarks such as bintree_search and pointer_chase, where scalarization is not possible. In practice, both techniques can often be combined synergistically to improve the execution time even further.

## 8. CONCLUSION AND FUTURE WORK

The automatic generation of application-specific microarchitectures for load value speculation can significantly improve the performance of memory-intensive programs. Future work will concentrate on reducing the area overhead of the required logic by packing storage and delay elements in a manner more amenable to efficient logic synthesis.

## REFERENCES

BUDIU, M. AND GOLDSTEIN, S. C. 2005. Inter-iteration scalar replacement in the presence of conditional control-flow. In *3rd Workshop on Optimizations for DSP and Embedded Systems*.

CALLAHAN, D., CARR, S., AND KENNEDY, K. 1990. Improving register allocation for subscripted variables. In *Proc. of the ACM SIGPLAN 1990 Conf. on Programming Language Design and Implementation - PLDI '90*. ACM Press, New York, New York, USA, 53–65.

CALLAHAN, T. J. 2002. Automatic Compilation of C for Hybrid Reconfigurable Architectures. Ph.D. thesis, University of California, Berkeley.

GÄDKE-LÜTJENS, H. 2011. Dynamic Scheduling in High-Level Compilation for Adaptive Computers. Ph.D. thesis, Technical University Braunschweig.

GÄDKE-LÜTJENS, H., THIELMANN, B., AND KOCH, A. 2010. A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation. *Intl. Conf. on Field Programmable Logic and Applications*, 475–482.

GONZÁLEZ, J. AND GONZÁLEZ, A. 1998. The Potential of Data Value Speculation to boost ILP. In *Proc. Intl. Conf. on Supercomputing*. ICS '98. ACM, New York, NY, USA, 21–28.

GONZÁLEZ, J. AND GONZÁLEZ, A. 1999. Limits of Instruction Level Parallelism with Data Value Speculation. In *Intl. Conf. on Vector and Parallel Processing*. VECPAR '98. 452–465.

HENNESSY, J. L. AND PATTERSON, D. A. 2003. *Computer Architecture: A Quantitative Approach* 3 Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

JOUPPI, N. P. 1990. 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*.

KAELI, D. AND YEW, P.-C. 2005. *Speculative Execution In High Performance Computer Architectures*. CRC Press, Inc.

KASPRZYK, N. AND KOCH, A. 2005. High-Level-Language Compilation for Reconfigurable Computers. In *ReCoSoC*.

KUMAR, S., PIRES, L., PONNUSWAMY, S., NANAVATI, C., GOLUSKY, J., VOJTA, M., WADI, S., PANDALAI, D., AND SPAANENBERG, H. 2000. A Benchmark Suite for Evaluating Configurable Computing Systems—Status, Reflections, and Future Directions. In *FPGA*. ACM, 126–134.

LANGE, H. AND KOCH, A. 2010. Architectures and Execution Models for Hardware/Software Compilation and their System-Level Realization. *IEEE Trans. on Computers*.

LANGE, H., WINK, T., AND KOCH, A. 2011. MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers. In *DATE*.

---

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


DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons systems. In *Proc. of the 30th annual ACM/IEEE Intl. Symp. on Microarchitecture*. MICRO 30. IEEE, 330–335.

LI, Y., CALLAHAN, T., DARNELL, E., HARR, R., KURKURE, U., AND STOCKWOOD, J. 2000. Hardware-software co-design of embedded reconfigurable architectures. In *DAC*. 507–512.

LIPASTI, M. H., WILKERSON, C. B., AND SHEN, J. P. 1996. Value Locality and Load Value Prediction. In *Proc. of the 7th intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. ASPLOS-VII. ACM, New York, NY, USA, 138–147.

MCNAIRY, C. AND SOLTIS, D. 2003. Itanium 2 Processor Microarchitecture. *IEEE Micro 23*, 44–55.

MOCK, M., VILLAMARIN, R., AND BAIOCCHI, J. 2005. An Empirical Study of Data Speculation Use on the Intel Itanium 2 Processor. In *Proc. Workshop on Interaction between Compilers and Computer Architectures*. IEEE Computer Society, Washington, DC, USA, 22–33.

SAM, N. B. AND BURTSCHER, M. 2005. On the Energy-Efficiency of Speculative Hardware. In *Proc. of the 2nd Conf. on Computing Frontiers*. CF '05. ACM, New York, NY, USA, 361–370.

THIELMANN, B., HUTHMANN, J., AND KOCH, A. 2011a. Evaluation of Speculative Execution Techniques for High-Level Language to Hardware Compilation. In *IEEE Proc. Intl. Workshop on Reconfigurable Communication-centric Systems-on-Chip*.

THIELMANN, B., HUTHMANN, J., AND KOCH, A. 2011b. PreCoRe - A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation. In *Proc. Intl. Conf. Field-Programmable Logic and Applications (FPL)*.

WANG, K. AND FRANKLIN, M. 1997. Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proc. of the 30th annual ACM/IEEE Intl. Symp. on Microarchitecture*. MICRO 30. IEEE Computer Society, Washington, DC, USA, 281–290.