EVALUATION OF SPECULATIVE EXECUTION TECHNIQUES FOR HIGH-LEVEL LANGUAGE TO HARDWARE COMPILATION

Benjamin Thielmann

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

Jens Huthmann, Andreas Koch

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

ABSTRACT

The PreCoRe approach allows the automatic generation of application-specific microarchitectures from C, thus supporting complex speculative execution on reconfigurable computers. In this work, we present the PreCoRe capability of using data-value speculation to reduce the latency of memory reads, as well as the lightweight extension of static datapath controllers to the dynamic replay of misspeculated operations. The experimental evaluation considers the performance / area impact of the approach and also discusses the individual effects of combining different speculation mechanisms.

I. INTRODUCTION

The combination of a software-programmable processor (SPP) and a reconfigurable compute unit (RCU) in an adaptive computing system (ACS) can provide computing performance at high efficiency: Computing kernels are realized as dedicated microarchitectures on the RCU, while the SPP implements just non-timing critical control operations.

However, in practical use, ACS architectures still occupy a niche. This is often due to the specialized expertise expected from the developer: Programming an ACS commonly requires experience in digital hardware design, computer architecture, and specialized programming languages and tool flows (Verilog/VHDL, place and route, simulation, ...).

To make the potential of ACSs available to more users, considerable research effort (e.g., [2], [15], [10]) has been expended on automatic compilers for such heterogeneous computers. These systems attempt to optimally distribute a high-level language program among the two different processing elements, synthesize the custom microarchitecture on the RCU, and generate the SPP/RCU interface in software and in hardware.

Traditionally, ACS compilers have concentrated on extracting instruction level parallelism (ILP) from their input programs to generate computing hardware on the RCU. However, with the improvements of the compiler technology, the classic memory latency bottleneck that has plagued SPPs for many years also materializes for the RCU: According to [9], 20% of a typical program's instructions are memory accesses, but they require up to 100x of the execution time of the non-memory (scalar) operations.

With this outlook, appropriate countermeasures should also be considered for the RCU when targeting an ACS. One of the most efficient solutions to increase the available memory bandwidth involves the switch from a single off-chip main memory to many local on-chip memories, all of which can be accessed in parallel [3]. However, the technique is not applicable to all algorithms and its use is as yet not fully automated. Less efficient, but more generally applicable, is the use of multi-port coherently cached memory systems, possibly backed by multiple parallel channels to main memory [13]. On the other hand, such systems perform best only if many *independent* memory accesses are present in the program. Otherwise, the inter-port coherency traffic will become the new bottleneck.

As an orthogonal approach, we propose the use of *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 (even in case of cache-misses or DRAM refreshes in the actual memory system). 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 re-executed (replayed) with the correct input values. When performing a replay, it is desirable limit the extent of the re-executed operations. Ideally, only the specific operations "poisoned" by the misspeculated values need to be replayed. However, implementing such a fine replay granularity requires complex hardware.

Our PreCoRe framework aims to add lightweight support for value speculation to custom-compiled microarchitectures. Instead of requiring dynamic scheduling at the level of individual operators (as discussed in [10]), PreCoRe predicts, commits, and replays at the granularity of datapath *stages* similar to those used in a statically scheduled microarchitecture.

While previous work [19] focused on the microarchitectural realization of the commit/fail and replay mechanisms, this text presents as new contributions the actual data value speculation techniques, the extension of statically scheduled datapath controllers to stage-based speculation, and a discussion of the interaction of different speculation effects. The latter include the degree of memory prefetching enabled by value speculation (in the specific form of address speculation). All of the presented mechanisms have been implemented in the hardware/software Co-Compiler Nymble, which is also used for the experimental evaluation in actual FPGA hardware.

II. RELATED WORK

Even assuming support for unlimited ILP in hardware, memory read data dependencies severely limit the degree of ILP achievable in practice to between tens to (at most) hundreds of instructions [8].

Lipasti et al. proposed load value speculation to resolve these read data dependencies speculatively [16], allowing computations to continue using the speculated values without waiting for the memory system to return the actual (possibly cached) memory contents.

The limits of value speculation to improve ILP have been discussed, e.g., in [5], [4]. A number of different approaches to speculatively determine the read value have been examined: History-based predictors such as as last-value prediction attempt to predict the next value by selecting it from a limited set of the previously read values. Stride-based predictors attempt to determine a new value by assuming a constant offset from the previously read value. Context value predictors look beyond individual values and attempt to supply entire sequences of values following a regular pattern (beyond just a single constant offset). Multiple schemes can be combined into hybrid predictors.

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. modified a compiler to force value speculation where possible on the Intel Itanium 2 CPU architecture [18]. Their scheme relied on hardware support in the form of the Itanium's Advanced Load Address Table (ALAT) [17]. However, the ALAT does not operate autonomously, but has to be explicitly controlled by software code. At best, they achieved speed-ups of 10% for read value speculation using this approach. However, they also observed slow-downs of up to 5% under adverse conditions.

III. DATA SPECULATION INFRASTRUCTURE

III-A. Integration with Memory System

The read value speculation capabilities of PreCoRe are implemented as transparent extensions to the MARC2 multi-port coherently-cached memory system [13], shown in Figure 1. The actual data prediction unit is inserted between the datapath-side read port and the MARC2-internal per-port cache. The remainder of MARC2 (intra- and inter-cache management, access to one or more physical memory channels) remains unmodified.

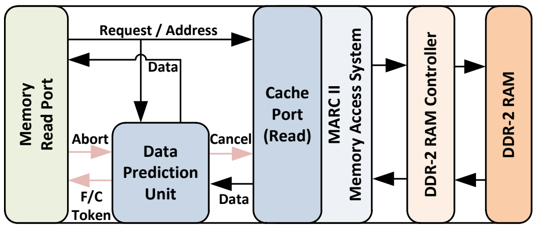

Fig. 1. MARC2 memory system extended with read value speculation

With this modification, read requests issued by the datapath on the RCU are simultaneously passed to the per-port cache and to the data prediction unit. Thus, the datapath will always receive a reply with 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.

III-B. Load Value Speculation

The efficiency of the scheme is highly dependent on the accuracy of the data prediction unit. Fortunately, data speculation techniques have been well explored in context of conventional processors [20], [1].

A key difference of these processor-centric approaches and our technique is due to the spatial-computing paradigm used in the custom-compiled accelerators. Processors generally have very few Load-Store Units (LSU), often only a single one. Thus, it is very simple to realize a predictor with a global view of the execution context: *All* loads pass through the single LSU where the predictor is located. This allows the predictor to make decisions similar to "with Load1 having returned 23, and Load2 having returned 42, I predict that Load3 will now return 2011". While this can be advantageous for some applications, in others an irregular value pattern for one of the load instructions can lead to an overall loss of prediction accuracy, since all predictions are made based on the global view. To alleviate this, the use of the Program Counter to differentiate between the value streams of different individual load instructions has been proposed, restricting the prediction context to a local view. Here, the reasoning would be similar to "Load1 did return 42 the last time, I predict it will return 23 this time; Load2 did return 8 the last time, I predict it will return 12 this time". With its separate cache ports for each individual read operation in the datapath, MARC2 also follows the spatial computing paradigm. Thus, the value predictors also have a purely local (per-port) view of the load value streams. In practice, the restricted local view will lead to longer training times before a predictor becomes accurate (it does not see the global context), but the predictor will be more resilient against irregular data patterns incoming via just some of the read operations.

Another aspect of using value speculation is how to train the predictors, specifically, when the underlying pattern database (on which future predictions are based) should be updated: Only once a speculation has been determined as correct/incorrect (which might take considerable time, since it could involve actually waiting for a read of main memory)? Or should the speculated values be assumed to be correct (and entered into the pattern database) until proven incorrect later? In PreCoRe, we have opted for the second choice, since a single inaccurate prediction will always lead to the re-execution of all subsequent read operations (and their internal predictions), then with pattern databases updated with the correct values. In contrast to the first approach, this will require the predictor hardware to be able to rollback the entire pattern database to the last completely correct state once a speculation has proven to be incorrect. Note that, even with the risk of re-execution, PreCoRe aims to never slow down the execution of the datapath over the non-speculative version (see Section VI for further discussion).

III-C. Predictor Architecture

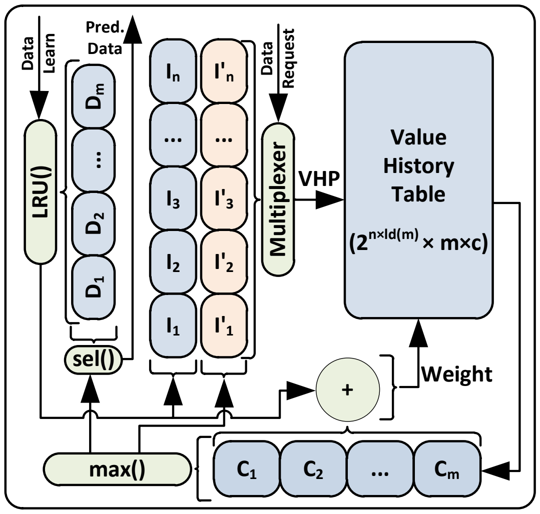

Fig. 2. Local history-based load value predictor

Figure 2 shows the basic architecture for 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 [20]. 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, 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). The complete PreCoRe value prediction unit consists of separate last-value and stride-predictors, operated in tournament mode (a predictor is trusted until it makes an incorrect prediction, then trust is switched to the other predictor). For simplicity, the following discussion will just refer to value prediction.

As 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. Thus, the VHT is accessed by a $n \log_2 m$-bit wide address and stores words of width mc bits.

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).

The operation of the resulting local history-based load value predictor is shown in Fig. 3 as an example. As parameters, we chose $m = 4$, $n = 6$, $c = 2$ (the default parameters for our current hardware implementation). In Figure 3.a, D stores 47,13,24,11 as the last $m = 4$ different values encountered. At the start of the execution, we assume here $I = I'$ (all values have been confirmed), with both master and shadow VHPs of (1,2,3,4,4,3) indicating that the sequence of the last $n = 6$ values was 47-13-24-11-11-24.

At this point in time, a new prediction request arrives. The shadow VHP is then used to access the VHT (Figure 3.a.I) to retrieve the likelihood of the current values in

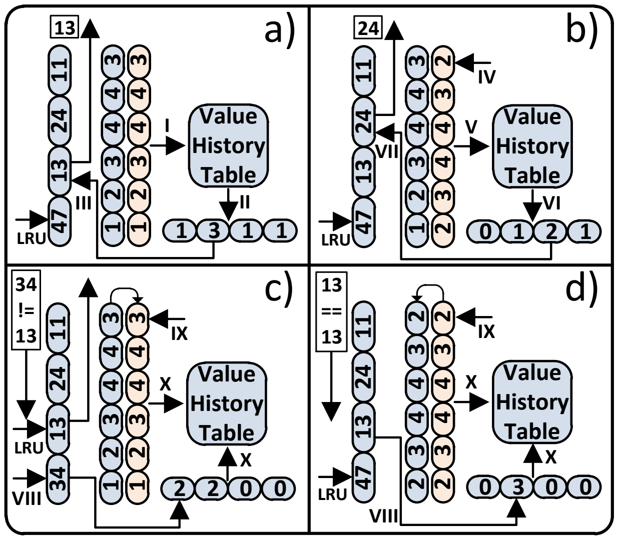

Fig. 3. Value predictor example

D (.a.II), expressed as the counter vector (1,3,1,1). Since $C_2 = 3$, the highest value (for a $c = 2$-bit counter), $D_2 = 13$ is predicted as the most likely value (.a.III) and output. Since the prediction always selects from a known value, the VHT does not need to be updated on a prediction request. However, the stored value patterns now have to be updated. Since the new value 13 is as-yet unconfirmed by the memory system, we update the *shadow* VHP I' to (2,3,4,4,3,2), adding the index of new value to the end of the sequence, but losing the oldest element. The next incoming prediction request (Figure 3.b.IV) then performs the next look-up using the updated shadow VHP (.b.V), retrieves the appropriate VHT entry (.b.VI), which then indicates by the highest value in C_3 that $D_3 = 24$ is the next most likely value (.b.VII).

Assume, however, that before that second request comes in, the memory system has provided the actual value for the first request (which lead to the prediction of 13). If that value was mispredicted (Figure 3.c), the correct value (here 34) is entered in the least-recently-used position of D, here D_4. (.c.VIII). Since the speculation was incorrect, the shadow VHP is reset by being overwritten with the master VHP (.c.IX), which is then used to access the appropriate VHT entry. That entry erroneously lead to the prediction of 13 and is now corrected from its old value (1,3,1,1) as follows: The counter for the index *matching* the correct value is increased (C_1 for $D_1 = 34$), all other counters are decreased using saturating arithmetic. The resulting counter vector (2,2,0,0) is then written back into the VHT (.c.X). Finally, both the shadow and master VHPs will be updated to (2,3,4,4,3,1), reflecting the sequence of confirmed values (not shown in the Figure). Had 13 be a correct prediction (.d.VIII), the shadow VHP would be copied to the master VHP (.d.IX), and C_2 in the appropriate VHT entry *increased* because $D_2 = 13$ was a correct prediction again (clipped by saturation at 3), with all other choices becoming more unlikely (and thus decreased). The resulting entry (0,3,0,0) is then written back into the VHT for the given shadow VHP (.d.X).

IV. STATICALLY SCHEDULED SPECULATION

IV-A. Nymble Compiler

Since we aim to automatically generate application-specific speculation mechanisms, we also have to consider the effects of the approach on the compiler. For this purpose, we have extended our hardware/software co-compiler Nymble, which itself is based on the Scale framework [7] [6]. The framework is used to perform front-end lexing/parsing and machine-independent optimizations, resulting in an intermediate representation (IR) of control-flow graphs (CFG) in Static Single Assignment (SSA) form.

For hardware synthesis, this SSA-CFG is then translated into a control data-flow graph (CDFG), from which both the controller and the datapath can be generated as synthesizable Verilog HDL. The results of some of the analysis passes are used to optimize the micro-architecture. E.g., Alias/Points-To analysis is used to assign memory accesses proven to be independent to separate coherency clusters in MARC2, thus reducing the pressure on the cache-coherency network [13].

Initially, Nymble generated pure statically scheduled hardware using a lightweight control sequencing scheme. The next section examines how this scheme can be extended with limited dynamic scheduling to support PreCoRe.

IV-B. Data Path Controller

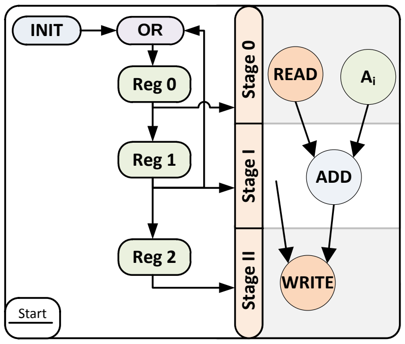

Fig. 4. Controller for non-speculative data path

Figure 4 shows the basic structure of the non-speculative datapath and controller. The datapath shown on the right consists of the actual hardware operators. Each has a `Start` control input signal which lets an operator evaluate

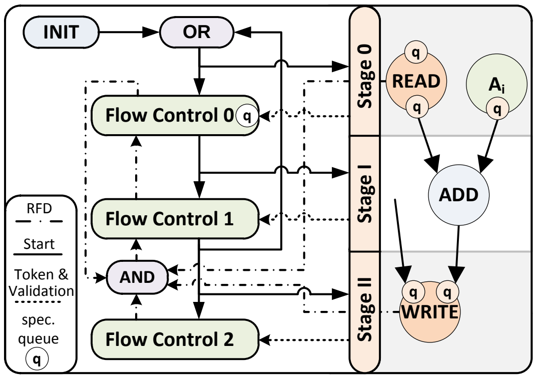

Fig. 5. Controller for speculative data path

its current input values and begin computing. As usual, all operators started in the same clock cycle are organized into a *stage*.

In the initial pure statically scheduled model, the execution of the stages is sequenced by a controller implemented as simple shift register: An external RCU-global start command (generally sent by the SPP) inserts the first '1' as `Start` token through the INIT block, leading to the activation of the first stage. The following stages are then activated in order when the '1' advances in subsequent clock cycles. The structure can express pipelined execution in loops by a back-edge going from stage II-1 to stage 0, where II is the initiation interval of the loop (in the figure, II=2). For variable-latency operators such as cached reads, we schedule for the expected latency of one cycle (cache hit). On a cache miss, the entire datapath is halted by stopping the datapath (deasserting clock enable) until the data is actually available on the outputs of the READ node.

In some regards, adding value speculation to the datapath actually simplifies the controller: Now, even memory reads become fixed-latency operations, since they will always output a result within a single cycle (either data available from a cache-hit or a speculated value). However, the need to replay operations executed using incorrectly speculated values complicates the speculation-supporting controller, shown in Figure 5.

To this end, the simple registers in the controller are replaced by so-called Flow Control nodes. In normal operation, they act identically to the simple shift register and delay their stage's `Start` token for a clock cycle before passing it on to the next stage. Two new capabilities are required by PreCoRe: First, since some operators in the datapath (among them memory reads and writes) are now fitted with input queues (described in detail in [19]), the `Start` token advances to the next Flow Control node(s) *only* if *all* the datapath stages controlled by the successor controller nodes have indicated that all of their contained operators have input queue space available by asserting Ready-for-Data (`RFD`). This operation is implemented by connecting the per-datapath stage `RFD` to the lower AND node in the controller part of Fig. 5. If even a single datapath operator lacks queue space, the `Start` token remains in the predecessor controller node, and execution pauses. The distribution of the `RFD` signal also has to take the loop-closing back-edge into account, which leads to a stage having two successor stages. In the figure, this is the case for Stage 1, which has Stage 0 and Stage 2 as successors. Thus, the `RFD` signals for both of these stages enter in the AND expression determining when a `Start` token may leave Stage 1.

Second, the `Start` tokens *themselves* are now also queued within the Flow Control nodes that control datapath stages affected by value speculation. These include the speculative read operations themselves, but also the write operations marking speculation boundaries; see [19] for details. The Flow Control node token queues are used in PreCoRe to replay operations that were performed on misspeculated values. In that case, the Flow Control node responsible for the datapath stage which emitted the incorrect data is advised to re-issue the `Start` token from its queue, causing the re-execution of subsequent operators with new values. A `Start` token is actually removed from a Flow Control queue only if the corresponding data path stage has confirmed that it executed on correct data (by the Commit/Validation mechanism of [19]). When the Flow Control nodes re-issue `Start` tokens from their queues, they also ensure that the tokens emerge at the same rate they arrived in (the loop II). Finally, with the Flow Control nodes now also having queues, a predecessor node is allowed to advance its `Start` token only if *all* successor nodes have space available in their token queues. This is achieved by back-pressure `RFD` signals in the controller (running upwards in Fig. 5), which also enter into the AND of all successor nodes' and stages' `RFD` signals. In the figure, Stage 1 may thus advance its `Start` token only if the operators in the datapath stages controlled by the successor nodes, as well as the successor control nodes themselves, have queue space available.

Nymble generates the appropriate extensions to the initial purely static controller in an application-specific manner: Queues and token transition conditions are inserted only where required for the current datapath. This avoids the overhead of a general-purpose approach, possibly even relying on finely-granular dynamic scheduling at the level of individual datapath operators [10].

V. TARGET ARCHITECTURE

The experimental evaluation of PreCoRe and its integration in the Nymble compiler have been performed on the ACE M5 adaptive computing system. Hardware-wise, the

ACE M5 consists of a Xilinx ML507 Virtex-5 FX development board. The on-chip PowerPC 440 processor acts as the SPP component of the ACS, while the remainder of the Virtex-5 FPGA fabric are used for the RCU. The main memory shared between SPP and RCU is implemented off-chip as DDR2-SDRAM DIMMs. However, a number of extensions have been made over the original Xilinx-provided environment.

First, memory accesses are are performed using the MARC2 memory access system [13], allowing the RCU direct access to the DDR2-SDRAM memory controller without having to go over the comparatively slow PLB bus. MARC2 provides each memory operation in the datapath with a dedicated cache port. The system can be configured by indicating which accesses occur to non-overlapping memory regions (determined by Alias/Points-To analysis in the Nymble compiler) to generate an application-specific sparse cache coherency communication network. Each coherency cluster may have an arbitrary number of read ports, but only a single write port. If more writes occur to address ranges potentially overlapping those of the read ports, multiple writes need to be sequentialized and issued through the single write port. An arbitrary number of parallel writes may be performed to non-overlapping ranges. In that case, the writes can be assigned to different coherency clusters. This arrangement allows PreCoRe to issue multiple memory operations in parallel per cycle.

Second, the ACE M5 platform runs under a heavily modified full-scale version of the Linux operating system that allows fast RCU-SPP signaling (up to 23x faster than even a kernel with real-time patches) as well as the use of virtual memory by the RCU using the AISLE technique [12]. The latter is important for allowing the SPP and RCU to freely pass pointers across the hardware/software boundary, a crucial capability when compiling from a pointer-intensive language such as C.

VI. EXPERIMENTAL RESULTS

VI-A. Performance Overview

This section shows the experimental results of applying the PreCoRe approach to a number of sample applications. Initially, the nature of the benchmarks will be described and the performance data presented. Then, selected aspects of the results will be discussed in greater detail, among them the interaction of different optimization effects.

Some of the kernels compiled to hardware accelerators on the RCU are selected from real application programs or benchmark suites, others have been designed synthetically to examine specific aspects of the technology.

gf_multiply is part of the Pegwit elliptic curve cryptography application, in itself part of MediaBench [14]. The kernel performs a Galois Field multiplication. The kernels versatility_fcdf22 and versatility_quantization are part of a wavelet image compression algorithm used in the Versatility benchmark of the Honeywell suite [11]. The sample median_filter_row realizes a luminance median filter, where blocks of 9 pixels are read and the median of the luminance is written to the center pixel. In addition to the usual approach of processing the image row-by-row, we also implemented a version median_filter_col proceeding column-by-column, to selectively disable the MARC2-internal cache line prefetching. This can be used to observe the impact of load-value speculation for cases when the cache efficiency deteriorates. Two other kernels also exercise difficult-to-cache cases by implementing pointer-chasing algorithms: pointer_chase traverses a linked list and increments the value of every second node. bintree_search searches for keys in a binary tree. To gauge the impact of data dependencies, we use two synthetic benchmarks: simple_read sums all values in an array, it has a loop-carried data dependency on the intermediate sum. array_add adds a scalar to all elements of an array, an operation without loop-carried dependencies.

For each kernel, Table I shows the simulated execution time, area requirements as well as maximum clock speeds. The synthesis and mapping tools used were Synopsys Synplify Premier DP 9.6.2 and Xilinx ISE 11.1. The default system clock frequency of our ML507-based platform is 100 MHz.

Due to increased complexity of the PreCoRe extensions, a datapath supporting speculative execution sometimes has a lower maximum clock speed than the original statically scheduled version. However, even with a clock frequency decrease of 1.5…11.4 MHz, many speculative datapaths still achieve the 100 MHz default clock rate, and thus do not actually suffer from the slower clock when considering system-level performance.

More serious is the area increase associated with speculation: Despite the lightweight stage-based approach avoiding operator-granularity scheduling, hardware kernels with speculation enabled require 1.45x…3.22x the number of slices of their non-speculative versions. It turns out that the overhead for the actual data prediction unit itself is limited to ca. 1000 LUTs per memory read operation, and the handshaking logic has an even smaller impact. Most of the overhead is due to the output queues inserted into the datapath by PreCoRe. When this issue is examined more closely, it turns out that the way Nymble currently creates the different storage elements (separate output queues and registers for pipeline balancing) is not processed efficiently by the logic synthesis tools: They do not take advantage of merging adjacent queues and registers for tighter hardware packing. For example, in pointer_chase, the number of registers increases by 6.91x for the speculative version, but 79.84% of them realize individual single-cycle pipeline balancing nodes that have not been merged into longer,

Kernel	FPGA Area				Max. Clock Freq.		Runtime				Comparison	
	#LUTs		#Registers		(MHz)		#Cycles		μs at max. freq.		Slices	Speed
	n.spec	spec	n.spec	spec	n.spec	spec	n.spec	spec	n.spec	spec	ovrhd.	-up
1 array_add	10141	14717	1246	2948	106.90	96.30	6194	3923	57.94	40.74	1.45x	1.42x
2 bintree_search	11129	19782	1570	5795	105.70	100.10	3497	3359	33.08	33.56	1.78x	0.99x
3 gf_multiply	11918	22790	1702	6459	102.80	101.30	2510	2482	24.50	24.42	1.91x	1.00x
4 median_filter_row	12895	28333	2458	9909	106.40	102.20	296409	115027	2785.80	1125.51	2.20x	2.48x
5 median_filter_col	12998	28391	2529	10325	106.00	100.30	1054736	666554	9950.34	6665.24	2.19x	1.50x
6 pointer_chase	11979	20637	1478	10208	106.40	98.90	4087	3650	38.41	36.91	1.72x	1.04x
7 simple_read	10241	14703	1484	3639	105.80	103.20	19615	14150	185.40	135.41	1.45x	1.37x
8 versatility_quant.	12351	39716	5390	18495	105.70	97.40	96771	50746	915.53	521.01	3.22x	1.76x
9 versatility_fcdf22	12055	32284	1889	9863	105.20	93.80	43633	20573	414.76	219.33	2.68x	1.89x

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

more efficient SRL blocks. In practice, the impact of the high demand for registers is somewhat limited when considering the slice-level area, as the slices holding the more complex speculation logic also supply more registers. However, future work will address this issue by pre-packing storage elements in the generated RTL Verilog.

Even with is current area increase and slower maximum clock frequency, our first PreCoRe implementation can lead to significant speed-ups of the sample kernels. When comparing PreCoRe kernels to the statically scheduled ones running at their maximum theoretical clock frequencies, PreCoRe achieves an average speed-up of 26.88%, going up to 2.48x for the median filter. The worst-case slow-down we observed was $< 1\%$ in case of bintree_search. Comparing against the real maximum clock of 100 MHz of the ACE M5, the average performance advantage increases to 30.1% for PreCoRe kernels, with a peak speed-up of 2.59x for median filter under these real conditions. With the maximum clock speed limited to 100 MHz, the use of PreCoRe does not slow-down any kernel.

VII. PERFORMANCE ANALYSIS

After examining the raw performance data in the prior section, we now discuss the interaction of two separate effects occurring when using read value speculation in PreCoRe.

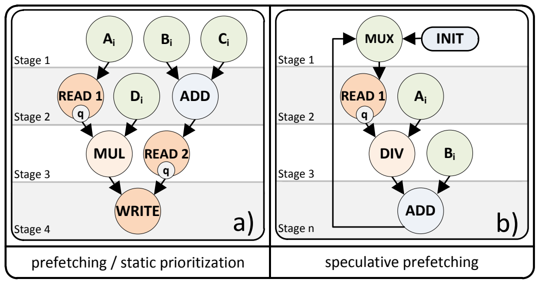

Fig. 6. Interaction of speculation with prefetching

VII-A. Prefetching

In the original, pure statically scheduled RCU shown in Section III, 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 has the effect of improved prefetching, even if the speculated read value turns out to be wrong.

Figure 6.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, pre-fetching data from a non-speculated 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 6.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 III-C. In practice, this might occur 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 Fig. 6.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 speed-up of 1.43x. Additionally enabling the value-predictor reduces the execution time further to 666,554 cycles, a total speed-up 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

Runtime	Hit Rate			Speed-Up
#Cycles	Total #Commits	Total #Fails	%	%
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

Table II. Execution time vs predictor accuracy for pointer_chase

responsible for most of the performance gain.

VII-B. 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 application has a basic structure similar to that shown for Figure 6.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 II 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.

VIII. CONCLUSION AND FUTURE WORK

With PreCoRe, we have introduced a general-purpose framework for extending simple statically scheduled compute units compiled from C with application-specific value speculation hardware. Specifically, this work has described the operation of the actual value speculation units and the microarchitectural changes required to the scheduling controller. We have shown the general speed-up potential, as well as discussed specific effects such as improved prefetching. Future work will deal with improving the area efficiency of the scheme as well as adding further dependency-resolution mechanisms such as store-load forwarding.

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

References

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

[2] T. J. Callahan. *Automatic Compilation of C for Hybrid Reconfigurable Architectures*. Ph.D. thesis, University of California, Berkeley, 2002.

[3] H. Gadke-Lutjens, B. Thielmann et al. A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation. *Intl. Conf. on Field Programmable Logic and Applications*, 0:475–482, 2010.

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

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

[6] S. C. Group. Scale - A scalable Compiler for Analytical Experiments, 2011.

[7] G.Weaver, B.Cahoon et al. Common Language Encoding Form (Clef) Design Document, Technical Report, U Boston, 1997.

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

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

[10] N. Kasprzyk and A. Koch. High-Level-Language Compilation for Reconfigurable Computers. In *ReCoSoC*. 2005.

[11] 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.

[12] H. Lange and A. Koch. Architectures and Execution Models for Hardware/Software Compilation and their System-Level Realization. *IEEE Trans. on Computers*, 2010.

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

[14] C. Lee, M. Potkonjak et al. 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, pp. 330–335. IEEE Computer Society, Washington, DC, USA, 1997.

[15] Y. Li, T. Callahan et al. Hardware-software co-design of embedded reconfigurable architectures. In *DAC*, pp. 507–512. 2000.

[16] M. H. Lipasti, C. B. Wilkerson et al. Value Locality and Load Value Prediction. In *Proc. of the 7th intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-VII, pp. 138–147. ACM, New York, NY, USA, 1996.

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

[18] 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.

[19] B. Thielmann, J. Huthmann et al. PreCoRe - A Token-Based Speculation Architecture for High-Level Language to Hardware Compilation, submitted to FPL 2011.

[20] K. Wang and M. Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proc. of the 30th annual ACM/IEEE intl. symp. on Microarchitecture*, MICRO 30, pp. 281–290. IEEE Computer Society, Washington, DC, USA, 1997.