An Efficient Memory Organization for High-ILP Inner Modem Baseband SDR Processors

Bjorn De Sutter
Ghent University, Belgium

Osman Allam, Praveen Raghavan, Roeland Vandebriel, Hans Cappelle, Tom Vander Aa and Bingfeng Mei
Interuniversity Micro-Electronics Center (IMEC), Belgium

Abstract. This paper presents a memory organization for SDR inner modem baseband processors that focus on exploiting ILP. This memory organization uses power-efficient, single-ported, interleaved scratch-pad memory banks to provide enough bandwidth to a high-ILP processors. A system of queues in the memory interface is used to resolve bank conflicts among the single-ported banks, and to spread long bursts of conflicting accesses to the same bank over time. Bank address rotation is used to spread long bursts of conflicting accesses over multiple banks. All proposed techniques have been implemented in hardware, and are evaluated for a number of different wireless communication standards. For the 11a|n benchmarks, the overhead of stall cycles resulting from unresolved bank conflicts can be reduced to below 2% with the proposed organization. For 3GPP-LTE, the most demanding wireless standard we evaluated, the overhead is reduced to less than 0.13%. This is achieved with little energy and area overhead, and without any bank-aware compiler support.

Keywords: interleaved memory, memory queues, software-defined radio, instruction-level parallelism, memory-level parallelism

1. Introduction

Digital signal processors (DSPs) for mobile software-defined radio (SDR) inner modem baseband processing, such as Silicon Hive's CSP2200(Hive, 2007), NXP's EVP (van Berkel et al., 2005) or IMEC's ADRES (Bougard et al., 2008b; Bougard et al., 2008a; Derudder et al., 2009), need to deliver very high performance at low energy consumption. Today, this is done by exploiting different types of low-level parallelism. When data-level parallelism (DLP) is available in an application, its exploitation by means of vector processing or SIMD (single-instruction, multiple data) processing is one of the most power-efficient techniques to optimize performance within single application threads. When available and exploitable, DLP offers the huge advantage that few wide ports to memory can suffice to provide the necessary data bandwidth, which is much more power-efficient than having many narrow ports. However, even if enough DLP is avaible, it is still hard to compile for, and hence

hard to exploit. This is particularly the case when data shuffling is needed, as when there are concurrent array accesses with different strides. For this reason, it remains an open question as to what extent DLP solutions will scale to future wireless standards. Using instruction-level parallelism (ILP) instead of DLP to exploit available parallelism is typically more flexible. Whereas DLP instructions typically perform the same operation on multiple data elements concurrently, ILP implementations enable the concurrent execution of different operations on multiple data elements. Typically, ILP processors are also easier to compiler for. Today Silicon Hive and ADRES have full compiler support, while EVP has no or very limited compiler support for its vector data path.

One downside of using ILP instead of DLP is that ILP is typically less power-efficient, however, because more instruction bits need to be fetched and decoded per data operation. On architectures such as SiliconHive and ADRES, this is compensated by the special loop modes that are used to execute loops. In these mode, code is fetched from dedicated memories that, unlike instruction caches, consume very little power. Furthermore, ILP can be combined with limited amounts of DLP. For example, the 4-way SIMD ADRES Inner Modem Baseband (Bougard et al., 2008b; Bougard et al., 2008a; Derudder et al., 2009) architecture with 16 issue slots proved to be a power-efficient competitive processor for running many wireless standards, including very demanding standards such as 3GPP and 802.16e. The ADRES architecture achieves its power-efficiency through two operation modes that are optimized for two types of code. First, a narrow, low-ILP VLIW mode with 3 issue slots executes non-kernel code. Secondly, a wide, high-ILP reconfigurable data flow mode with 16 issue slots accelerates inner loops that operate on arrays of data. Of course, this high-ILP mode requires a high memory bandwidth to feed the numerous issue slots with data. As the DLP is limited, this bandwidth can only be provided through multiple memory ports.

Designing a memory organization for a single wireless standard that consists of a few inner loops, a.k.a. kernels, is relatively simple. For high-ILP SDR processors, the design is more complicated. First, there are more different kernels and hence more different access patterns to support. Furthermore, those patterns are often less optimized and less regular. The lack of optimization results from code reuse over multiple standards. While reusing code is beneficial for limiting the overall code footprint, the reuse of a kernel prohibits its specialization or optimization for any single execution context. The lack of regularity results from the code generation techniques used in high-ILP compilers (Mei et al., 2003; Park et al., 2008; Friedman et al., 2009; De Sutter et al., 2008; Oh

et al., 2009). These compilers usually generate software-pipelined code in which loads and stores from many iterations are mixed in irregular ways that are most often not easily predictable at compile time. These properties also warrant the use of multiple ports to memory.

With respect to that memory itself, it is well-known that scratch-pad memories, when used well, consume less power than caches(Baert et al., 2008; Wehmeyer and Marwedel, 2006). Furthermore, single-ported memories consume less power than multi-ported memories. This implies that, if we can compile our code in such a way that it accesses all 2^n banks in a memory organization exactly once every cycle, then a scratch-pad memory with 2^n single-ported banks and 2^n load/store units will be a very power-efficient organization that can provide precisely the amount of throughput required. In practice, however, it is not feasible to generate such code for irregular, unoptimized data access patterns. Instead it will occur that multiple load/store operations try to access the same single-ported bank together, thus causing so-called bank conflicts. Because most DSPs feature static code schedules with blocking loads, such bank conflicts stall the processor until all outstanding accesses are resolved. The resulting stalls can impose significant limitations on the achievable performance and energy consumption.

This paper presents a novel, hardware-supported conflict resolution mechanism to avoid such stalls and to support high memory bandwidth while still enabling the use of power-efficient single-ported scratch-pad memory banks. The core idea of the proposed conflict resolution mechanism for high-ILP VLIW-like processors such as the ADRES SDR Inner Modem Baseband processor, is to spread same-bank access bursts over time and over multiple banks. The mechanism consists of the following components:

1. **Data memory queues** exploit additional execution cycles given to issued load/store operations in software-pipelined loops to resolve bank conflicts. Thus, *small bursts* of simultanuous accesses to the same bank are spread over a number of cycles. Because the longer individual latencies of load operations are not problematic in software-pipelined loops, this do not prohibit efficient loop schedules.

2. **Load/store reordering** further spreads bursts of interleaved loads and stores to the same bank by delaying the execution of stores. This is possible because loops in SDR applications typically operate on streaming data. Since data written to memory inside such a loop cannot influence any later read operations in that loop, delaying the store operations does not change the behavior, even if this delaying involves the reordering of loads and stores. As such, load/store

reordering helps in spreading *small bursts* of loads and stores to the same bank over time.

3. **Bank access rotation** changes the assignment of memory locations to interleaved memory banks to make sure that the stride of the interleaving is not a power of two. Thus, *long bursts* of accesses to the same non-rotated bank, which resulted from access patterns with strides that are powers of two, will access multiple rotated banks instead.

Together, these techniques reduce the number of conflicts that need to be resolved with processor stalls to negligible amounts, while requiring little additional power consumption and no specific compiler support. This is demonstrated by compiling several wireless standards programmed in ANSI C for an Inner Modem Baseband ADRES SDR processor, and by performing gate-level simulations of the processor and memory organization.

The main contribution of this paper is the combination of the above techniques (which are not new by themselves), the novel implementation of the data memory queue controller by means of scoreboards, and the evaluation of a real hardware implementation for the latest wireless standards, including even the emerging but not yet fully standardized 3GPP-LTE. This evaluation demonstrates that previously proposed dynamic techniques for addressing interleaved memory banks are not needed in practice for these wireless standards. Instead static techniques such as load/store reordering and static bank access rotation implemented on top of memory queues suffice. Of course the implementation presented in this paper can be useful in other domains too, such as video coding. Whether or not the static approach presented here suffices in those contexts is out of scope of this paper.

The structure of the paper is as follows. Section 2 describes the context in which a memory organization for a high-ILP inner modem baseband SDR processor needs to operate. It discusses properties of applications and of the VLIW-like architecture to which the memory is attached. Section 3 presents our three solutions conceptually, and Section 4 presents a hardware implementation. This implementation is evaluated in Section 5. Section 6 discusses related work, and Section 7 draws conclusions.

2. Processor and Application Context

The inner modem baseband wireless algorithms that we want to run on a SDR processor have a number of important properties, which

define part of the context for which our memory organization should be optimized. The other part of the context is defined by our choice for a high ILP VLIW-like processor with a scratch-pad L1 data memory. In this section, we discuss this context.

2.1. Basic Assumptions

Both the transmitter and the receiver of all standards are streaming applications in which a number of consecutive loops, hereafter called kernels, operate on arrays of data that are stored in the scratch-pad memory. We want to execute those loops on a high-ILP VLIW-like processor or accelerator, as is done, for example, on state-of-the-art commercial processors such as SiliconHive (Hive, 2007). The SiliconHive CSP2200 features a narrow VLIW mode for non-kernel code, and a wide VLIW mode for executing software-pipelined kernels. We focus on the wide VLIW mode in this paper because that mode requires most memory bandwidth.

Before looking at the memory hierachy and related properties, we need to clarify a number of limitations of our wide VLIW mode. First of all, we should note that the energy and area constraints of mobile devices limit the amount of ILP that an architecture can exploit. In order to balance performance, energy, area, and available ILP in the code, the number of ALUs/multipliers on an architecture is always limited. As a result, some kernels will be CPU-bound because they offer more ILP than can be exploited by the number of available ALUs and multipliers. Other kernels will not be CPU-bound because data dependencies prohibit exploiting the amount of ILP that the architecture can sustain. Furthermore, we assume that load operations are blocking and that the code schedules are static, assuming fixed latencies for load operations. If the memory organization cannot handle a load operation in time, the whole processor will stall during which the memory gets more time to handle the operation. Also, in our argumentation for high bandwidth requirements, we take for granted that we are working with an architecture that supports predication for all operations. However, predication is by no means a requirement for our proposed solution. Finally, all numbers presented in this paper were obtained for a 64-bit processor (32-bit integer logic and 4x16 bits SIMD) which supports only 8, 16 and 32-bit memory accesses. We come back to this latter limitation in Section 4.7.

2.2. Computation Bandwidth and Memory Bandwidth

On such a high-ILP processor with predicated, blocking loads, the computation bandwidth provided by the number of ALUs/multipliers

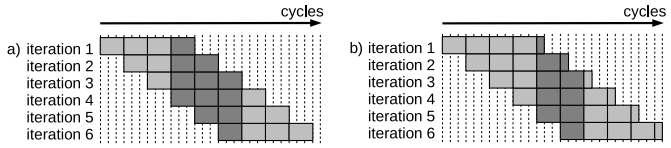

Figure 1. Execution traces of a software-pipelined loop compiled for two target architectures. Both versions have an II of three, but version a) has a shorter schedule because the assumed latency of the load operation is architecture a) is lower than on architecture b).

should be balanced with the memory bandwidth provided by the number of load/store units. In order to optimize performance and utilization, our memory organization should ideally be able to handle one memory access per cycle per load/store unit to feed data to the available ALUs and multipliers, without needing to introduce stall cycles. This means that in a perfectly balanced architecture, the average kernel will be as memory-bound as it will be CPU-bound.

By simulating perfect memories that can handle all memory accesses without needing to insert stalls, we observed that a high-ILP mode with 16 issue slots, of which 4 can also be used to issue memory accesses, is a good balance. With such architectures, average IPCs over 10 can be obtained easily at what we believe to be a sweetspot for performance and energy efficiency (Bougard et al., 2008b; Bougard et al., 2008a; Novo et al., 2008; Derudder et al., 2009). As an example, consider our 11a WLAN SISO transmitter. It spends 1141 duty cycles (i.e. non-stall cycles) in the high-ILP mode, during which it performs 3144 memory accesses out of 3741 scheduled ones; 597 accesses had false predicates. This means that the load/store units had a utilization of $3741/1141/4 = 0.82$ on average over all kernels. While the 11a WLAN SISO transmitter is an overly simple example for SDR processing, similar behavior is observed in the more complex SDR standards that we will evaluate later in this paper.

To interpret this number correctly, consider the software-pipelined execution of a loop depicted in Figure 1a. With software-pipelining (Rau, 1995; Lam, 1988) the loop body is split in pipeline stages. In this case, there are four pipeline stages, shown as rectangles from left to right. The loop is scheduled in such a way that each stage takes three cycles, and a new iteration can be started every three cycles. The loop is said to have an initiation interval (II) of three. In the first part of the loop execution, which is called the prologue and which is indicated in light grey on the left, stages become active progressively. During the steady state, which is indicated in darker grey, all stages are active together, for different iterations. At the end of the loop, which is called the

epilogue and which is also colored light gray, stages gradually become disactivated again as fewer and fewer iterations need to be finished. In this paper, we assume kernel-only loops, i.e. loops in which the prologue and epilogue code are part of the wide VLIW mode. In such kernel-only loops, the prologues and epilogues are implemented by means of so-called staging predicates, i.e. predicates that disable stages when they should not be active. Because of these staging predicates, every loop involves memory accesses that are predicated, even if the original source code of the loop did not include any conditional statements. At least during the prologues and epilogues these predicates will evaluate to false a number of times, which explains why there are more scheduled memory accesses than actually executed ones. During the steady-state, all staging predicates evaluate to true, so during the steady-state, all scheduled memory accesses are actually executed. In short, the peak memory bandwidth is required during steady-state, which is about 0.82 * 4 = 3.3 accesses per cycle.

Given these numbers and alike numbers for other wireless standards, our task is to design of a memory organization that can handle close to four memory accesses per cycle in steady-state without inserting too many stall cycles, and with as low as possible power consumption.

2.3. Scratch-pad Memory Organization and Access Patterns

Regarding scratch-pad memories, we have already noted that single-ported memories consume much less energy per access than multi-ported memories. Single-ported memories are therefore preferable if the same performance can be obtained with them as with multi-port memories. Obviously, when using single-ported memories, there should be at least as many as there are load/store units issuing memory accesses in the steady state of kernels. So ideally we want a hierarchy with just as many single-ported memories as there are load/store units. From here on, we will use the term memory *banks* to denote the single-ported memories that constitute one bigger memory space. The best organization of these banks depends on the kernel's memory access patterns.

2.3.1. *Bank Conflicts and Load Latencies*

We assume that a compiler cannot (always) control which banks will be accessed by which load/store units. One of the reasons is that some kernels access arrays in data-dependent ways that a compiler cannot analyze. Also, some kernels are used in many standards, and hence in many different contexts, some of which feature different memory layouts

of the accessed arrays. Consequently, all load/store units need to be connected to all memory banks, for example by means of a crossbar. And it needs to be foreseen that bank conflicts will occur. A bank conflict occurs when two load/store units want to access the same bank in the same cycle. Even if on average all banks are accessed the same number of times, there may be simultaneous accesses to the same bank from multiple load/store units.

Since each bank has only one port, it can handle only one access per cycle. The other accesses to the same bank have to be handled in later cycles. Either stall cycles need to be inserted, or a mechanism needs to be foreseen through which the processor does not expect the accessed data to be available immediately, in which case it does not need to stall. Such a mechanism can, for example, exploit additional time obtained by increasing the latency of load instructions in the architecture.

Figure 1b shows the same loop as in Figure 1a, but in this case it is scheduled assuming higher latencies of load operations. The effect of higher latencies is of course longer schedules, and in this case the schedule of one iteration has gone from 12 to 13 cycles, which is an increase in schedule length of about 8.3%. For the whole loop, however the performance loss is much lower than 8.3%. Basically, it is not because the schedule of a single iteration has become longer that the compiler has become unable to find a schedule with the same II. Figure 1b depict such a schedule, and one can see that the total loop execution time is only increased with one cycle, which is only a 3% cycle increase. To verify this on real kernels, we've scheduled several FFT versions with different latencies of load operations, ranging from 3 to 12 cycles. The accumulated duty cycle counts of these kernels are depicted in Figure 2. It can be seen that the number of duty cycles barely increases, from 593 to 681, when the latency of loads is increased from 3 up to 12. This indicates that higher load operation latencies might be able to provide ample time for a bank-conflict resolution mechanism. Of course, the increase in duty cycles resulting from higher load latencies should then be compensated by a reduction in stall cycles. As in so many situations, a trade-off between throughput and latency needs to be found.

2.3.2. *Memory Access Patterns and Interleaving*

A kernel's memory access pattern determines whether or not the kernel will cause many bank conflicts. Some important kernels access only one array, which they read and *update in place*. Most often this update in place happens because the input and output array are overlayed onto the same memory locations to reduce the memory footprint. If that whole array is placed in a single memory bank, the bandwidth to the array will be limited to one access per cycle. To avoid this it suffices not

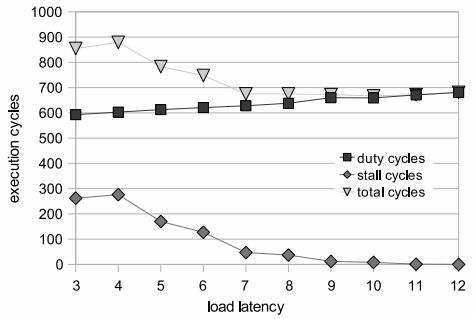

Figure 2. Duty cycles and stall cycles when using different sizes of unified queues, corresponding to different load operation latencies, to access non-rotated interleaved banks.

bank 0	bank 1	bank 2	bank 3
0x00	0x04	0x08	0x0c
0x10	0x14	0x18	0x1c
0x20	0x24	0x28	0x2c
0x30	0x34	0x38	0x3c
0x40	0x44	0x48	0x4c
...	...	...	...

Figure 3. A simple interleaved 32-bit memory organization. The numbers inside each 32-bit wide bank denote the addresses of the words stored in them.

to assign contiguous memory regions to banks, but to assign memory to banks in an interleaved fashion. Figure 3 depicts a simple interleaved bank assignment. To avoid the need for complex integer divisions in the memory hierarchy, its number of banks should be limited to powers of two. That is one of the reasons why 4 load/store units is the best number for architectures with 16 ALUs/multipliers.

Of course, the effectiveness of an interleaving scheme depends on the strides with which arrays are indexed. If the strides occurring in a loop are bad, the whole loop might access only one bank. This happens, e.g., when a stride of 0x10 is used to access a single array stored in the interleaved memory of Figure 3. For such cases we use the term *long burst* of conflicting accesses to a memory bank. But even when the stride in a loop looks perfect for some interleaving scheme, there might be occasional conflicts or even sequences of conflicts. We will term those *short bursts*. One reason why there might be conflicts even with good strides is that there is no strict relationship between the order in which the memory accesses occur in the source code of an application and the order in which they will be performed in the loop schedule. This follows from the fact that loads and stores might be reordered by the compiler, and by the fact that software-pipelined schedules are generated, in which code corresponding to different iterations in the source code is executed simultaneously. In our current version, the compiler (Mei

et al., 2003; De Sutter et al., 2008) schedules the code completely bank-unaware.[1] As such, even if the strides might look alright in the source code, there might still be many conflicting simultaneous accesses in the scheduled code. Of course, if the strides look alright in the source code for some interleaved bank assignment, this implies that at least on average the accesses will be spread over multiple banks. So in that case, the only problem the memory organization needs to tackle is that of accidental, short bursts of accesses to the same bank.

2.3.3. *Aliasing and Memory Dependencies*
Most if not all kernels in the streaming applications that are run on an SDR inner modem baseband processor do not contain read-after-write or write-after-write dependencies. In each kernel, one or more arrays are read, computations are performed on the read data, and the results are written to other arrays that will be consumed later on by other kernels. Sometimes arrays being written may also be read, as is the case with the aforementioned update in place. But then each read from a memory location will always happen before all writes to that location, so there again are no read-after-write or write-after-write dependencies.

A compiler can easily check whether or not a loop contains read-after-write or write-after-write dependencies through memory, for example by means of loop analysis techniques based on the polyhedral model analysis (Bastoul et al., 2003). Alternatively, if no such compiler analysis is available in a compiler, the programmer can indicate the presence or lack of these dependencies in code annotations that are presented to the compiler.

Once the compiler can detect, through analysis or through annotations, that the above types of dependencies are not present in a kernel, it knows that the write operations in the kernel can be delayed and reordered without any problem.

3. Memory Queues and Bank Rotation

Given the context described in the previous section, we have to design an interface between a processor with 4 load/store units, and 4 single-ported interleaved memory banks. This interface needs to reduce the

[1] Finding valid schedules on our wide ADRES, which is in fact a coarse-grained reconfigurable array of 16 ALUs and 13 register files with a sparse interconnect, is very complex. To find those schedules, we currently rely on simulated-annealing, which is quite slow. Making the compiler bank-aware might make it even slower. Also, as we will see in the evaluation section, we provide a solution based on rather cheap hardware that makes bank-aware code generation unnecessary altogether.

cycles	instr.	issue	accesses BANK 0	accesses BANK 1	retire
1) no queue, no rotation					
0	0	r0, r1	r0		
1	stall		r1		r0, r1
2) unified queue, no rotation					
0	0	r0, r1	r0		
1	1		r1		r0, r1
3) unified queue, no rotation					
0	0	r0, w0	r0		
1	1	r1, r2	w0		r0, w0
2	2		r1		
3	stall		r2		r1,r2
4) separate queues, no rotation					
0	0	r0, w0	r0		
1	1	r1, r2	r1		r0
2	2		r2		r1,r2
3	3		w0		w0
5) no queues, no rotation					
0	0	r0, r1, r2	r0		
1	stall		r1		
2	stall		r2		r0,r1,r2
3	1	r3	r3		r3
6) no queues, rotation					
0	0	r0, r1, r2	r0	r1	
1	stall		r2		r0,r1,r2
2	1	r3		r3	r3
7) separate queues, rotation					
0	0	r0, r1, r2	r0	r1	
1	1	r3	r2	r3	r0,r1,r2
2	2				r3

Figure 4. The execution of schedules on seven memory organizations. The first column indicates the clock cycles. The second column shows the VLIW instructions executed. The third column depicts the memory accesses issued in each cycle. The fourth and fifth column indicate the actual memory accesses being performed, and the last column which instructions retire.

number of stall cycles as much as possible. Stalls will occur when there are short (accidental) bursts of accesses to the same bank or when the number of accesses, over a longer period of time, is not balanced over the available banks.

To avoid both types of stalls, we will rely on three techniques. First, we will increase load latencies to enable the implementation of a queueing mechanism that can resolve conflicts by spreading them over time. Next, these queues will be extended with write buffers that can delay conflicting write operations, thus spreading short bursts to the same bank over even more time. Finally, bank rotation will ensure that long bursts of accesses to a single (non-rotated) bank are spread over multiple banks, thus avoiding long bursts to a singe bank altogether.

3.1. Data Memory Queues

When the latency of load operations is increased, data memory queues can be inserted in between the processor and the memory banks to resolve bank conflicts. These queues basically spread short bursts of

simultanuous accesses to the same bank over a number of cycles. Because the longer individual latencies of load operations are not that detrimental in software-pipelined loops as long as they remain small enough, as discussed in Section 2.3.1, the efficiency of the loop schedules is not hampered significantly.

To illustrate how this works, consider the first two execution traces in Figure 4. In all traces for organizations without queues (organizations 1, 5, and 6 in Figure 4), we assume for the sake of simplicity that memory accesses have a latency of one cycle: they need to retire in the cycle in which they are issued. Although this is not realistic, as memories do have latency, this assumption does not change the concepts illustrated here. On the organizations with queues (organizations 2, 3, 4, and 7 in Figure 4) we assume that the latency is increased to two cycles, i.e. the result only has to become available at the end of the next cycle. We also assume that all accesses issued in the third column go to bank 0 of a non-rotated interleaved memory as depicted in Figure 3.

Now consider two accesses (r0 and r1) that are issued to bank 0 in cycle 0 of organization 1 in Figure 4, which does not have queues. Because bank 0 can only handle one of the two accesses, the second one is delayed to a stall cycle. At the end of that stall cycle, both memory accesses retire, after which the processor can continue executing the second instruction.

On organization 2 of Figure 4 with queues and higher latency, the read operations issued in cycle 0 only need to retire in cycle 1. So bank 0 can handle r0 in cycle 0, and r1 in cycle 1, after which both operations retire in cycle 1. They retire in the same cycle as in the organization without queues, but in this case, the processor did not need to be stalled. Assuming that the compiler was able to place other operations in the remaining slots in that cycle, more operations will have been executed in the two cycles.

For a benchmark consisting of several different FFTs, the reduction in stall cycles by using data memory queues and increased load latencies is depicted in Figure 2. The smallest load latencies of 3 and 4 cycles that our processor (and its pipeline implementation) can support, are too small to enable the insertion of queues. Consequently, there is no conflict resolution at those latencies, which results in a very high numbers of stalls. With higher latencies, we can insert data memory queues that can resolve more and more conflicts in the additional time they get from the increased latencies. Thus, the number of stall cycles goes down when the load latency increases. Above 7 cycles of latency, however, the number of stalls does not decrease significantly anymore. In fact, above 7 cycles of latency, the reduction in stall cycles can only

compensate the increase in duty cycles. This indicates that at least for this small benchmark, a load latency of 7 cycles is optimal.

We should note that in architectures in which the low-ILP and high-ILP mode share the memory interface, as in the one used in our evaluation, the queues need to be disabled during low-ILP mode. In code that is not software-pipelined, increasing the latencies of load operations comes with a much bigger degradation in IPC (instructions per cycle), so there we want the shortest possible latency. The disabling of the queues is discussed in Section 4.5. Here, we limit the discussion to clarifying that disabling the queues in low-ILP mode is most often not problematic. The most frequently occurring sequences of memory accesses in non-loop code are the spilling and filling of callee-saved registers to the stack upon entry to and exit from a called procedure. Typically these registers are spilled to consecutive addresses, which results in few conflicts in interleaved banks.

3.2. Write Buffers

Organization 3 in Figure 4 features a queue and increased latency. A read operation and a write operation are issued in cycle 0 of the schedule and two read operations are issued in cycle 1, all to the same bank. r1 and r2 need to retire in cycle 2. Obviously, handling four operations in three cycles is impossible, so again the memory needs to stall the processor for one cycle.

In organization 4, the memory interface stores the read and write operations in separate buffers, and it gives priority to the read operations. Assuming that the write is independent of any subsequent reads, the memory interface can now handle the three read operations in cycles 0, 1, and 2, and the write operation in cycle 3. Again, more instructions will have been executed in the same amount of cycles. Just like the original unified queues can spread conflicting access over time without stalling the processor, so can the separate write buffers. They simply offer more spreading capability.

It should be clear that the write buffers we use to delay writes in favor of reads have nothing in common with the well-know store buffers found on out-of-order superscalar processors. There write buffers store data to be written to memory in order to forward it to consecutive read operations. Those read operations can then be handled without having to wait until all outstanding writes accesses are committed to the actual memories. In our case, no forwarding is done whatsoever. This makes the implementation much cheaper, as we will see in Section 4.

As it can happen that some writes are delayed in favor of reads throughout the whole execution of a kernel, some writes might still

be outstanding when a kernel finishes. To handle those writes, special hardware needs to be foreseen that can introduce additional stalls at the end of a kernel's execution. In practice, an accumulation of outstanding writes towards the kernel's end will occur infrequentely. One reason is that the peak bandwidth to memory is only required during a kernel's steady state. In the epilogue, more and more memory accesses are disabled by their staging predicates, which will almost certainly free slots to handle outstanding writes. Secondly, because of the bank rotation proposed in the next section, there only occur short bursts of conflicts anyway. So over the whole execution of a kernel, no significant amount of outstanding writes will have accumulated.

Finally, we should note that the use of separate write buffers needs to be software-controlled to guarantee the correct execution of kernels for which the compiler cannot assume that there are no read-after-write or write-after-write dependencies. This software control by means of a flag set or reset upon entry to a kernel does not come at the expense of significant additional hardware.

3.3. Bank Access Rotation

Memory queues and separate write buffers give the memory the ability to work around short bursts of bank conflicts. What remains to be handled are the long bursts. Such bursts occur, for example, when arrays are accessed with a bad stride. For example, consider the following C code fragment:

```
int* pm;
for (...) {
  y0 = *(pm + 0*4);     y1 = *(pm + 1*4);
  y2 = *(pm + 2*4);     y3 = *(pm + 3*4);
  y4 = *(pm + 4*4);     y5 = *(pm + 5*4);
  y6 = *(pm + 6*4);     y7 = *(pm + 7*4);
  ...
}
```

In this fragment, all accesses to the array `pm` occur at indices that are multiples of 4. Since integers have a width of 4 bytes, all the offsets in the binary code will be multiples of 16, so in the interleaved bank organization of Figure 3, all these accesses to `pm` will access the same bank.

The problem of such accesses to the same bank is also illustrated in the schedule of organization 5 in Figure 4. Three read operations are issued in the first cycle of the schedule, and one more is issued in the second cycle. In between, the memory has to stall the processor

for two cycles because the first three accesses are to the same bank. The schedule of organization 6 illustrates that the number of stalls goes down if rotation is used, but still one stall cycle is required. Now suppose the interleaving of the banks is different (in our case this will be rotated), such that the read operations r1 and r3 no longer access the same bank as r0 and r2. The corresponding execution trace is depicted in organization 7 of Figure 4. In this case, no more stall cycles are needed.

The problem to solve here is related to the problem of row-major storage and column-major storage of matrices and the order in which the matrices are traversed, and to the problem of determining an array layout given the access stride patterns. Many data layout and address generation optimizations have been proposed in the past to optimize accesses to arrays, especially in the context of systems with distributed memories such as single cores with caches, multicores with non-shared caches, MPSoC systems with scratch-pad memories, etc. (Catthoor et al., 2002; Kandemir et al., 1999; Wehmeyer and Marwedel, 2006). The transformations applied and the hardware support proposed usually aim at reducing the number of conflicts for memory banks and at improving the locality for caches by carefully scheduling accesses in the correct order. In most proposed techniques, transformations on layout or access order are fine-tuned, parametrized or set for each kernel separately. This is because typically some kernels are best with row-major layout or order, while others are better off with column-major. The same holds for kernels that access arrays with different strides.

In our case, the compiler is assumed to be bank-unaware, and many kernels are memory-bound. So the compiler cannot avoid conflicts by reordering or spreading the accesses, and it cannot generate a setting for the parameters for a programmable memory organization. Instead we have to use a memory organization that works well for both row-major and column-major order, and for different strides. Furthermore, we don't care about improving locality. We assume that the programmer has written his program such that all data accessed is available in the scratch-pad memory. So we don't care about the order in which operations are executed or about the order in which banks are accessed. Finally, we can rely on the queues and write buffers to resolve short bursts of conflicting accesses. So here we only care about long bursts.

These observations imply that we can reorganize the memory banks differently without having to care about turning good access patterns into bad ones. We only have to make sure that any access pattern that was on average spread over all original banks (as organized in Figure 3) remains spread over the reorganized banks. The queues and

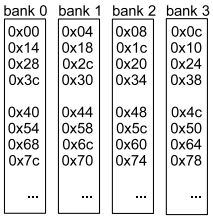

Figure 5. A rotated interleaved scratch-pad memory.

buffers will then resolve all accidental short burst conflicts that result from the reorganization.

Following this reasoning, we propose the bank organization of Figure 5. In this organization, it is as if the second row has been rotated to the left over one position, the third row has been rotated to the left over two positions, and the fourth row has been rotated to the left over 3 positions. This rotation is repeated throughout the whole scratch-pad memory. With this memory organization, it can easily be seen that the accesses to `pm` in the above code fragment are now nicely spread over all banks, as if all bank accesses got randomized.

An important aspect of this rotated bank assignment is its hardware cost. Suppose we have a 32-bit address space. We denote an address $b_{31}...b_0$. In the original interleaved 32-bit bank assignment of Figure 3, the bank is determined by the bits b_3b_2. Within the bank, the row is determined by $b_{31}...b_4$, and within a row, the bytes addressed are determined by b_1b_0. In the rotated interleaving, the same bits $b_{31}...b_4$ and b_1b_0 determine the row that is accessed and the bytes within that row. Only the bits that determine the bank change. Instead of b_3b_2, it is now the two bit value $b_5b_4+b_3b_2$ that determines the bank accessed. We call such a data interleaving *single rotation*. Clearly very little logic will be required to implement the bank rotation. More complex rotations can also be envisioned. For example, in the evaluation section we will also study *multiple rotation*, in which the bank accessed is determined by the two bit value $b_{13}b_{12} + b_{11}b_{10} + b_9b_8 + b_7b_6 + b_5b_4 + b_3b_2$.

We should note that in architectures in which a low-ILP mode shares the memory interface with a high-ILP mode, the use of rotated banks can occasionally introduce new conflicts in the low-ILP mode. As there is no conflict resolution of short bursts in that mode, the numbers of stalls in it may increase. Overall, we have found that this rarely happens, and only in insignificant quantities that certainly do not undo the gains obtained in high-ILP mode.

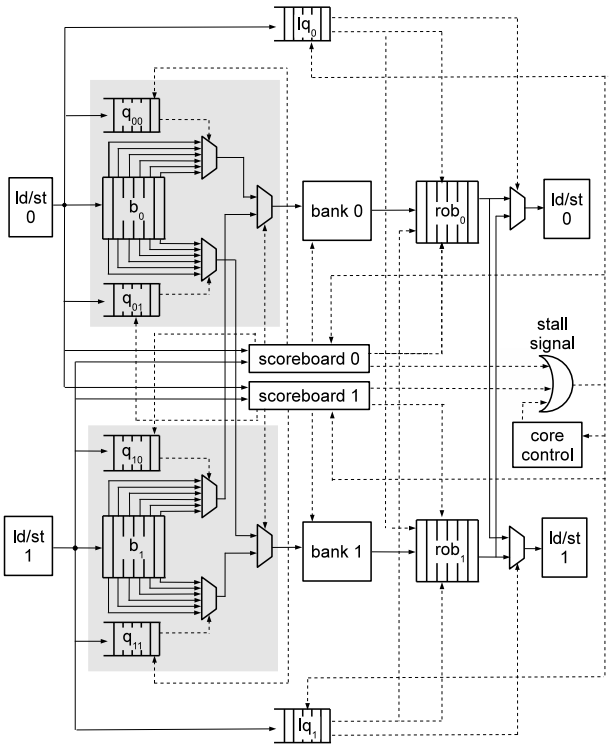

Figure 6. Overview of our hardware implementation of the unified data queues.

4. Hardware Implementation

4.1. Overview

Figure 6 depicts an overview of our unified data memory queue implementation for the simplified case of two load/store units and two memory banks. Queue enabling or disabling signals are not included to simplify the figure. Solid lines indicate data, while dashed lines indicate control. Its components are as follows.

– The *load/store units* at the left of the figure model the pipeline stages in which memory accesses are issued. The load/store units on the right of the figure model the writeback stages. Since we are working in the context of a VLIW-like processor with blocking loads with fixed latencies, a fixed number of cycles needs to pass between the issuing of a memory access and its writeback.

– Each memory bank y has a *reorder buffer* rob_y. Data that has been read from a bank is stored in this buffer until the correct cycle

arrives to send the data to the writeback stage of the processor pipeline.

- Each load/store unit x has a *latency queue* lq_x. This queue remembers when values that have been read from memory need to be send to the writeback stage, and uses this information to control the reorder buffers.
- Each memory bank y has a *scoreboard* $scoreboard_y$. This keeps track of the number of issued and outstanding memory accesses for its bank, for all load/store units. When it detects that it cannot handle all outstanding accesses in time, it will raise a stall signal.
- The *core controller* controls all stall signals. It is notified about stalls by the scoreboards when a memory access cannot be handled in time. There might also be other reasons for the processor to stall, for example, when a miss occurs in the instruction cache. Whatever the reason is for which the processor stalls, the memory interface needs to know about this to keep track of the time left to handle outstanding memory accesses. So the stall signal is sent to the latency queues and the scoreboard.
- Each load/store unit x has a *load/store buffer* b_x that stores the data of all outstanding memory accesses of the load/store unit. This information includes the address to be accessed, the type of operation (load or store), and, in the case of store instructions, the data to be written.
- Per pair (x, y) of load/store unit x and of memory bank y, there is one *queue* q_{xy} that stores pointers to the data in b_x for outstanding accesses to bank y.

The areas in grey form so-called single-input, multiple output (SIMO) queues. There is one SIMO per load/store unit. This unit pushes memory accesses into the SIMO, and the scoreboards pop them when they can be handled by their memory banks. In our hardware implementation, the reorder buffers are implemented in a similar way, but we have not drawn their internals to simplify the figure. In the case of reorder buffers, the memory bank pushes data that was read from that bank, and the latency queues pop data. The use of our SIMO structures has several advantages that are also known in the domain of network switches. These advantages will become clear in the next section.

4.2. Operation

Suppose load/store unit x issues a load operation. The unit first determines the bank y that will be accessed, using any of the address bit sequences discussed in Section 3.3. It then pushes information into the appropriate blocks as follows:

- The value y is pushed into lq_x. The depth of this queue equals the latency of a load operation, and this queue is popped every non-stall cycle. So the value y will be popped exactly when the data read for this memory access has to enter the writeback stage of load/store unit x, at which point it will be used as a selector for the multiplexor in front of the load/store unit's writeback stage.

- $scoreboard_y$ is notified that unit x wants to access its bank, and stores this information. How this information is stored and handled, is discussed in detail in Section 4.3.

- The read address is stored in buffer b_x, say at location i. This address will be obtained from this table when $scoreboard_y$ decides to pass this access to bank y. How this decision is made is also discussed in Section 4.3.

- The value i is pushed into queue q_{xy}, from which it will be popped when $scoreboard_y$ decides to pass this access to bank y, i.e. when this i reaches the front of q_{xy}, and when $scoreboard_y$ decides that it is x's turn to access bank y.

For store operations, about the same happens. But in this case, a special NO_DATA-value is pushed into the latency queue to indicate that there will be no data to pass to the writeback stage, and the data to be stored is stored in b_x, together with the store address. In a cycle in which no memory access is issued at all on unit x, because no such operation was scheduled or because its predicate evaluated to false, the NIL-value is also pushed into the latency queue.

When $scoreboard_y$ decides that a memory access issued from unit x can be handled by its bank y, that scoreboard pops the location i at which the access is stored in b_x from q_{xy}. It then uses i to retrieve the memory address, the type of operation (read or write), and, in the case of a store, the data from b_x. This retrieval is all done by means of the multiplexors in the SIMO of unit x. The retrieved data is sent to the memory bank to actually perform the access. Also, location i is freed in b_x. For a write operation, the handling finishes here.

For a read operation, the data that was read from bank y is pushed into rob_y, together with the value x of the load/store unit to which

the value has to be returned. This value comes from $scoreboard_y$. As already indicated, the reorder buffers internally look exactly like the SIMOs. They have one internal buffer in which the read data is stored at location j, and two queues (one for each load/store unit). The scoreboard pushes the value j into the appropriate queue.

When the x corresponding to this access gets popped from lq_x, the j will be at the front of its queue in rob_y. Using that popped j, the data at location j is retrieved from the internal buffer of rob_y and passed to the writeback stage of unit x. The entry j is freed in the internal buffer, which finalizes this memory access.

The advantages of our SIMO structure over other structures that would only consist of simple queues are that (1) multiple items can be popped from the SIMO simultanuously; (2) there are no head-of-line blockings of accesses from the same unit to different banks; instead such accesses can be reordered without any problem; and (3) its control is rather simple.

4.3. Scoreboards

The function of the scoreboards is threefold.

1. Each cycle, each $scoreboard_y$ has to decide which outstanding access is retrieved from the load/store unit SIMOs to be executed on bank y.

2. If the executed operation is a load operation, its result has to be stored in reorder buffer rob_y.

3. Finally, if the scoreboard detects that not all outstanding accesses can be handled in time, it has to raise the stall signal. When the stall signal is actually raised, no more memory accesses will be issued by the load/store units, and no data needs to be passed from the reorder buffers to the writeback stages. During the stalls, the scoreboard keeps selecting and performing bank accesses, and it keeps pushing read values in the reorder buffers, where they will wait until the stall signal is reset, after which the data will be passed to the writeback stages.

Figure 7 illustrates how a fictitious scoreboard for four load/store units works. In the scoreboard, there is one row per load/store unit, and a number of columns that is determined by the optimal load/store latency as discussed in Section 4.4. In this example, we assume that the load instruction latency is high enough to enable the memory interface to handle all accesses in time. In other words, no stalls need to be inserted. Ten consecutive cycles are displayed. In each cycle, a number

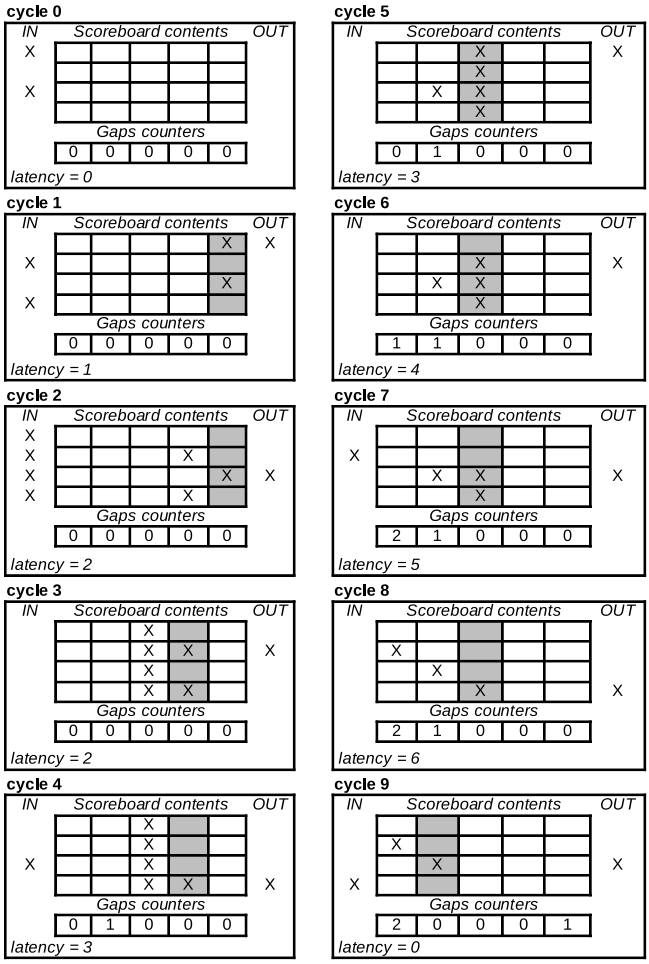

Figure 7. Operation of the scoreboard.

of accesses are issued and become visible to the scoreboard. The units issuing the accesses in a cycle are marked with X in the corresponding row under the heading IN, and the load/store unit of which an access is actually being passed to the bank is marked with an X under the heading OUT. Outstanding accesses are marked with Xs in the scoreboard table. The column marked in grey corresponds to the cycle of which the scoreboard is currently handling accesses to the memory bank.

Accesses enter the scoreboard one cycle after they show up at its input. The scoreboard can then start handling them, unless it still has to handle accesses from previous cycles. It handles one access per cycle, and proceeds from top to bottom, from right to left (and then rotating

at the end). This guarantees that all accesses to its bank are performed in order. Hence no memory dependencies can be violated.

If during some non-stall cycle no new accesses are issued to a bank, the gaps counter of the next free column is incremented by one. So every non-stall cycle, either the next empty column is filled with Xs, or that column's gaps counter is incremented.

Every time the scoreboard passes the last access in some column to the memory, this implies that it has finished all accesses that were issued in the same cycle. That column becomes empty, and the scoreboard is ready to start handling accesses in the next column. At that point, the gaps counter of that column is reset to 0, which models the fact that we also finished executing all loads and stores (being none) that were issued in the cycles in which this gaps counter was incremented.

The latency value specified in each cycle in Figure 7, to which we will refer as l_s, equals the sum of all gaps counters added to the number of columns with Xs in them. Basically this is a counter that is incremented every non-stall cycle, and that is decremented whenever all memory accesses of a cycle have been handled. In other words, the latency value specifies the number of non-stall cycles that have passed in the processor between the current cycle and the cycle in which the access was issued that is currently being passed to memory. When this latency value passes some threshold, the scoreboard knows that it will not be able the handle outstanding accesses in time. At that point, the scoreboard will raise the stall signal.

4.4. Latencies and Sizes

Previous sections described the operation of the memory interface and its exploitation of higher load operation latencies. Now we discuss the relation between the fixed latencies of load operations at the instruction set architecture (ISA) level and the sizes of the tables in our hardware implementation.

Our memory interface needs to support a fixed number of cycles from the moment a load/store unit issues a memory access to the moment the reorder buffer passes the result back to the write-back stage. We call this number *maximum latency*, or ml. The relation of this ml to the latency of a load operation as described in the ISA depends entirely on the pipeline structure of the processor's data path. At least the latency according to the ISA should be higher than ml, but how much it should be higher depends on the processor's pipeline. So in this section, we will determine all queue and buffer sizes in terms of the architecture-independant ml.

The tape-out samples (Derudder et al., 2009) of our prototype processor that includes this memory hierachy can operate at 400 MHz. It is fabricated using the 90 nm standard cell TSMC GP process technology. To reach this frequency, at least one cycle is spent accessing the memory bank. Also, at least one cycle is needed for an access to pass the queues and one cycle to pass the reorder buffer. This means that an access can spend any number of cycles c_q in the queues with $1 \leq c_q < ml - 2$, after which it spends 1 cycle on actually accessing the memory bank, and $ml - c_q - 1$ cycles in the reorder buffer.

To compute the sizes of the queues, buffers and tables, we need to know the exact points at which stall cycles will be inserted. They need to be inserted if the scoreboard detects that it will not be able to handle all outstanding requests in time. Since it takes at least two cycles out of ml cycles to perform an access and to pass the result through the reorder buffer, a stall should be inserted when the latency l_s as computed in the scoreboard (see Section 4.3) becomes higher than $ml-2$. So when $l_s > ml-2$, we know that stalls will have to be inserted, which needs to be done somewhere in the next two processor cycles. The exact point can be chosen because it takes at least two cycles for an access to get from the scoreboard to the writeback stage anyway. In our implementation, we have chosen to insert the stall one cycle after the cycle in which the scoreboard detects it needs to insert a stall. This was done because otherwise the combinatorial paths become too long to reach 400 MHz.

With that implementation, the scoreboard needs to be able to store at least the accesses of $ml-1$ cycles to be able to detect the need for stall cycles in the case all gap counters are zero. Furthermore, since the stall is delayed one more cycle, one more cycle can issue memory accesses before the stall actually happens. So in total ml columns are needed in the scoreboard. The number of rows is the number of load/store units.

Load operations issued on a load/store unit need to be finished in ml cycles. During those ml cycles, at most one new access per cycle can enter the unit's SIMO. And one additional access can enter the queues in between the detection of the need for a stall cycle and the insertion of the stall cycle. So in total, at most $ml + 1$ accesses from a load/store unit can be in flight. When this worst-case scenario happens, at least one of the accesses in flight is actually accessing the bank, and one more is in the reorder buffer. So worst case, the queues have to hold $ml + 1 - 2 = ml - 1$ accesses. That is their maximum size, and it corresponds to instructions spending anything in between 1 and $ml - 2$ (+1 in case of a stall) in the SIMO. It is the possibility to execute accesses at any time within this range that allows our whole organization to resolve conflicts. Of course, it is not necessary to make

the queues and buffer of the SIMO $ml - 1$ deep. Smaller depths can also be used, to save on area and on power, but then there will be less freedom to resolve conflicts, and hence more stall cycles will have to be inserted.

The size of the reorder buffers equals the number of load/store units plus one. This is because in the worst case, all units can have issued a load operation to the same bank in the same cycle, and then all these operations must have their results passed from the same reorder buffer to all writeback stages of all load/store units. Plus, because of the delay in the stall signal, one more read value can have entered the reorder buffer.

Finally, the latency queues can be implemented as simple pipelines of depth ml, since every non-stall cycle these queues get pushed and popped exactly once.

4.5. Disabling the Queues

At the end of Section 3.1, we argued that it should be possible to disable the queueing mechanism in order to achieve shorter latencies for non-pipelined code. In hardware, this can be achieved easily by making two values parametrizable, and by changing the values when switching between low-ILP mode for non-pipelined code and high-ILP mode for pipelined code. In the low-ILP a value of 1 instead of the value $ml - 2$ should be used when checking l_s for the need to insert stalls. This will ensure that every access leaves the queues in front of the memory bank after at most one (non-stall) cycle. Secondly, the depth of the latency queues should be set to 3. This ensures that all performed memory read operations have their data sent back to the processor pipeline as soon as it comes out of the bank.

4.6. Write Buffers and Rotating banks

The previous sections described the unified memory queues. We will not present the separate read queues and write buffers in detail. The SIMOs of the load/store units are duplicated (one for loads and one for stores), and the scoreboards are extended with counters to account for outstanding stores. Simple counters suffice for this because the writes can be reordered when the write buffers are enabled. If they are disabled (under control of the software), both loads and stores go to the load SIMO, through which they will still be executed in order.

Because writes in the separate write buffers can be delayed virtually indefinitely, write buffers can be made larger than $ml + 1 - 2$, unlike the sizes of all other elements, which are limited by ml as discussed in the previous section.

This implementation needs to work in two software-controlled modes. In one "unified" mode, one of the duplicated queues functions as a unified queue to execute code that contains read-after-write dependencies or write-after-write dependencies in order. In the other "reodering" mode, the duplicated queues operate as separate read queues and write buffers. To optimize area and power comsumption, the separate read queue of the "reordering" mode is best implemented in the queue that is not used in "unified" mode. This way, that queue does not need to provide space to store values to be written to memory. Instead it only needs to store addresses from which to read.

To implement rotating banks, it suffices to adapt the computation of the bank value y inside the load/store units.

4.7. Extensions

So far, we have assumed that the memory banks are scratch-pad memories. For the above hardware to work, this is not necessary however. With little modifications, other memories like caches can replace the scratch-pad memories. Also, the hardware is easily extensible to different numbers of load/store units or different numbers of banks. When no interleaving is required, the number of banks also does not need to be a power of two. It then suffices to adapt the current bank computation in the load/store as discussed in Section 3.3.

Another possible extension is to use 64-bit memory ports. So far, we have not done this because we have a significant number of kernels that cannot exploit 64-bit ports. They would underuse 64-bit ports, and hence become less power-efficient. Furthermore, our high-ILP mode shares the load/store units with the low-ILP mode for accessing the scratch-pad memory. In that low-ILP mode, there are more accesses with a stride of four bytes, such as the spilling and filling code previously mentioned in Section 3.2. Having 64-bit memory banks that are 64-bit wide would hence result in more conflicts in the low-ILP mode.

Another extension that is straightforward is the addition of ports to external memory. For example, it is easy to construct a system with four scratch-pad memory banks that cover the address range 0x0000-0xffff, and to add a fifth "bank" that is not a real bank but, e.g., an AHB master port to a DMA controller. Such a setup is in fact the one we implemented in our evaluation prototype ADRES processor used in the evaluation in the next section. Similarly, an external controlled DMA-controller can be connected as an AHB slave port that functions like an additional load/store unit.

4.8. Discussion

The scoreboard implementation proposed in this paper has a number of advantages over other implementations that only contain queues. The main advantage of using a scoreboard is that all information regarding the scheduling of memory accesses (to a bank) is centralized. In designs without a scoreboard, the scheduling information must be stored in the queues themselves as tags. This information then flows through pipeline together with the accesses, which complicates the decision logic.

The centralized scoreboard also facilitates extensions such as separate write buffers that can be enabled or disabled by the programmer, and other extensions such as extra ports for DMA controllers. Without any changes to the decision logic, to the design of the queues themselves, or to their interconnect, the scoreboard-based design also supports architectures with separate FUs for loading and for storing, or with more or less load/store units than banks. Supporting on and off chip memories, or multiple non-interleaved on-chip memories, poses no problem either. Even the adaption to use caches instead of scratch-pad memories, which means that the memory accesses themselves no longer have a fixed single-cycle latency, only requires relatively simple changes to the scoreboards, and only to the scoreboards.

5. Evaluation

5.1. The Evaluation Architecture

The results in Figure 2 have shown that it is clearly necessary to include some form of conflict resolution in the interface to the memory banks. They also showed that increasing the latency of load operations beyond a certain threshold number of cycles to give more freedom to the conflict resolution is not useful. Above that threshold, the decrease in the number of stall cycles is compensated by the increase in duty cycles. In this evaluation, we therefore focus on the conflict resolution methods for a fixed load instruction latency of 7 cycles as seen in the ISA of our target architecture. On that architecture, this corresponds to a fixed maximum latency or ml (see Section 4.4) of 5 cycles.

Some other important properties of our architecture are as follows. The whole architecture has a 64-bit integer data path to support 4-way 16-bit SIMD (single-instruction-multiple-data) operations and regular 32-bit integer operations. No floating-point operations are supported. This architecture is an instance of the ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) architecture template (Mei et al., 2004), that features a low-ILP VLIW processor and a

high-ILP coarse-grained reconfigurable array (CGRA). Inner loops are mapped onto the CGRA using modulo-scheduling, while the other code is mapped onto the VLIW processor. In our SDR implementation, the low-ILP mode is a 3-issue VLIW machine, in which all issue slots can issue memory accesses. It shares its main register file with the high-ILP CGRA mode to pass data between the two modes. Both modes execute exclusively: when an inner loop is entered, its CGRA code is invoked from within the VLIW code, and when the loop is exited, control is transferred back to the VLIW code. The VLIW processor also shares the memory interface with the CGRA, but in VLIW mode, the queues are disabled, which corresponds to a load instruction latency of 5 cycles (in the ISA) instead of 7.

The high-ILP CGRA mode features 16 issue slots. Of the 16 functional units, four can perform memory accesses besides regular ALU operations. All units can also perform integer 16-bit (SIMD) multiplications, and one unit can perform 24-bit divisions. The SIMD operations supported include simple parallel operations like parallel addition, subtraction, and shifting in saturated arithmetic, as well as complex fixed-point multiplication. All operations in the ADRES VLIW mode and in its CGRA mode can be predicated. Predication by means of hyperblock formation (Mahlke et al., 1992) allows the compiler to remove control flow from inner loop bodies, and thus enables the mapping of complex loop bodies onto the CGRA, on which control flow (apart from loop iteration) is not supported to make the very wide 16-issue instruction fetching more power-efficient.

All experiments performed in this paper are based on gate-level simulations of a full ADRES core (before placement and routing), which includes testing hardware, DMA controllers, a debug interface, etc. In short, all required features are present to make this core operate as a slave inner modem baseband processor in a multi-core SDR System-on-Chip. Such an SDR platform, containing two ADRES cores (featuring unified queues and non rotated bank assignment to a 4-bank scratch-pad memory of 64kB, and a CGRA memory of 128 736-bit instructions) was taped-out in Q2 08 and is currently being tested. That SDR platform uses 90nm TSMC general-purpose technology, in which the ADRES cores operate at 400MHz and occupies $6mm^2$ of die area. More detailed simulation experiments performed on that SDR platform, after placement and routing, have shown that a single SDR ADRES core can process the IEEE802.11n 20MHz 2x2 MIMO OFDM baseband code in real-time, consuming 220mW on average (310mW in CGRA mode, 75mW in VLIW mode), of which the typical leakage is 12.5mW (25mW at 65C). This proves that the processor used for this evaluation is competitive in terms of performance and energy

consumption. For more detailed info on our Inner Modem Baseband ADRES processor, we refer to (Bougard et al., 2008a; Bougard et al., 2008b; Derudder et al., 2009).

Compared to the taped-out ADRES cores, we increased the total level-1 scratch-pad memory size of the ADRES core used in this evaluation to 256 kB to make it fit our current implementation of the 3GPP-LTE standard. We also increased the size of the instruction memory for CGRA mode to 256 instructions, and we recently added a level-0 loop buffer in front of it to limit its power consumption. The instruction cache for the VLIW mode remains identical to the one in the taped-out cores, being a direct-mapped cache of 32 kB. Furthermore, we also added the optional bank rotation and separate write buffers. Before performing any measurements, we verified that none of the added features or changed memory sizes resulted in changes to the critical paths of the processor. As such, we can guarantee that the 400 MHz clock speed can still be obtained with all of the proposed conflict resolution schemes and with the increased memory sizes.

5.2. The Wireless Standards

An ADRES processor is programmed in sequential ANSI C. Our proprietary compiler extracts the loops from the application automatically, and maps them onto the CGRA. This mapping is done by means of modulo-scheduling to exploit the amount of available ILP as much as possible. SIMD is not extracted automatically by the compiler however. SIMD operations must be programmed manually by inserting so-called intrinsics in the C source code. Furthermore, our prototype compiler does not yet apply complex loop transformations such as loop unrolling to increase the amount of available ILP. For more information on the compiler, we refer to (Mei et al., 2003; De Sutter et al., 2008). Alternative compiler code generation techniques for architectures like ours exist (Park et al., 2008; Friedman et al., 2009; Oh et al., 2009). Their treatment of memory bank assignments and memory addressing schemes is not different from the one in our compiler.

The 6 standards and modes we have implemented are listed in Table I. Table II lists the kernels in each benchmark. Each of these applications was simulated at gate-level on the 12 organizations of the memory subsystem presented in Table III. Each organization is a specific combination of no rotation, single rotation or multiple rotations with unified read and write buffers or with separate write buffers of depths 4, 5 and 6. This means that at most 4, 5, or 6 write operations can be outstanding for any bank at any time. Table IV presents some basic results of this mapping exercise for the SDR inner modem base-

Table I. Standards used in this evaluation.

shorthand	standard and mode
11n Tx	IEEE 802.11n, SDM 2x2 20MHz, 64-QAM constellation, transmitter
11n Rx	IEEE 802.11n, SDM 2x2 20MHz, 64-QAM constellation, receiver
11a Tx	IEEE 802.11a SISO WLAN, transmitter
11a Rx	IEEE 802.11a SISO WLAN, receiver
3GPP-LTE Tx	3GPP-LTE SISO, 5MHz, 16-QAM constellation, transmitter
3GPP-LTE Rx	3GPP-LTE SISO, 5MHz, 16-QAM constellation, receiver

Table II. Kernels in the different standards.

shorthand	kernels
11n Tx	2x modulator, 2xIFFT, 2x add cyclic prefix
11n Rx	channel estimation, fine frequency synchronization, 2xFFT, cross-correlation, carrier frequency offset compensation, I and Q complex sample imbalance estimation, linear minimum mean square error detection, 2x demapping
11a Tx	modulator, IFFT, add cyclic prefix
11a Rx	channel estimation, fine frequency synchronization, FFT, cross-correlation, carrier frequency offset compensation, I and Q complex sample imbalance estimation, demapping
3GPP-LTE Tx	modulator, IFFT, add cyclic prefix
3GPP-LTE Rx	channel estimation, fine frequency synchronization, FFT, cross-correlation, carrier frequency offset compensation, I and Q complex sample imbalance estimation, demapping, interpolation

band processing standards of Tables I and II. For each benchmark, the fraction of the total execution time spent in the high-ILP CGRA mode is presented (as obtained on a basic memory interface with unified queues and no rotation) as well as the instructions-per-cycle (IPC) that are obtained in the high-ILP CGRA mode (counting only duty cycles). As can be seen, pretty high IPCs are obtained. We also included the power consumption estimates obtained using gate-level simulation, averaged over both the low-ILP VLIW mode and the high-ILP CGRA mode. These are lower than the aforementioned 220mW because of the level-0 loop buffer we added to the configuration memory. The power consumption is clearly related to the fraction of time spent in the CGRA mode.

To assess the pressure on the memory organization, Figure 8 presents the distribution of the number of memory accesses that are performed per duty cycle in the CGRA mode. Four accesses are performed in between 6.9% and 37.4% of the CGRA duty cycles, depending on the benchmark. Three accesses are performed in between 13.4% and 35.2% of the CGRA duty cycles. From these numbers, it becomes clear that we indeed need more than 2 ports to memory to deliver high performance. For completeness, Figure 9 shows the distribution of memory accesses issued per duty cycle for the low-ILP mode. Since this is a 3-issue VLIW processor, the maximum number of accesses in a single cycle is three.

Table III. Properties of the 12 memory interfaces organizations.

shorthand	read and write Queues	rotation
UQ - noROT	unified queues	none
UQ - sROT	unified queues	single
UQ - mROT	unified queues	multiple
WB4 - noROT	separate write buffers of depth 4	none
WB5 - noROT	separate write buffers of depth 5	none
WB6 - noROT	separate write buffers of depth 6	none
WB4 - sROT	separate write buffers of depth 4	single
WB5 - sROT	separate write buffers of depth 5	single
WB6 - sROT	separate write buffers of depth 6	single
WB4 - mROT	separate write buffers of depth 4	multiple
WB5 - mROT	separate write buffers of depth 5	multiple
WB6 - mROT	separate write buffers of depth 6	multiple

Table IV. Exploitation of the high-ILP mode.

standard	fraction high-ILP mode	IPC in high-ILP mode	power estimation before layout
11n Tx	53%	9.99	135mW
11n Rx	63%	10.60	152mW
11a Tx	53%	11.05	127mW
11a Rx	64%	10.37	141mW
3GPP-LTE Tx	96%	8.76	171mW
3GPP-LTE Rx	98%	11.30	196mW

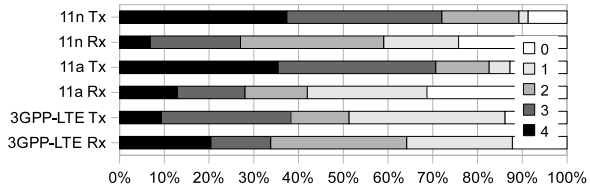

Figure 8. Distribution of the number of performed memory accesses per cycle in high-ILP CGRA mode.

Clearly, the memory bandwidth requirements of the VLIW mode are much lower.

Note that the CGRA mode is the only mode in which the different queues and buffer organizations are actually enabled. In VLIW mode, they are simply disabled. By contrast, any type of rotation that is applied in the CGRA mode needs to be applied in VLIW mode as well. This follows from the fact that the two modes share a single level-1 scratch-pad memory, in which they may access the same data.

5.3. Performance Results

Figure 10 presents the fraction of the total execution time spent in stalls resulting from bank conflicts for each of the 12 memory interface organizations. For all benchmarks, we immediately observe that consid-

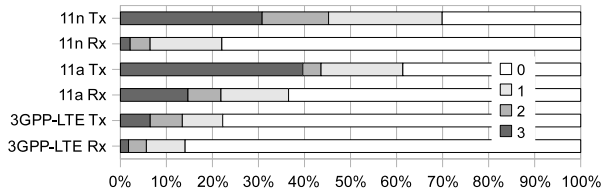

Figure 9. Distribution of the number of performed memory accesses per cycle in low-ILP VLIW mode.

erable fractions of the execution time are spent in stall cycles because of unresolved bank conflicts when only the basic queues are used.

Furthermore, the behavior clearly differs from benchmark to benchmark. With the limited number of benchmarks implemented so far, we notice that for less advanced standards, such as 11a and 11n, the relative number of stalls in a receiver is typically lower than in a transmitter, at least for the simpler memory organizations. This is to be expected, as those transmitters perform much less complex computations per transmitted byte than their receivers. Thus, the pressure on memory is much higher in the transmitters, as could be observed in Figure 8 as well. The reason why these transmitters spend relatively little time in CGRA mode, while still pressuring memory quite a bit in VLIW mode, is that they involve irregular data reordering operations that are not executed in loops. Instead those reordering operations consists of sequence of loads and stores executed in VLIW mode.

The 3GPP-LTE Tx and Rx behave differently. Both their pressure on memory in the CGRA mode is not that high. But on the other hand the 3GPP-LTE Tx and Rx spend almost all of their time in that mode. So in the end the 3GPP-LTE Tx and Rx still have a considerable amount of stall cycles, at least when the simplest conflict resolution is used.

Fortunately the number of stall cycles in all benchmarks can be reduced to below 2% by implementing separate write buffers and bank rotation. For the 3GPP-LTE Tx and Rx, the number of stall cycles is even below 0.13%. This shows that the proposed extensions are definitely useful to increase performance. On average, multiple bank rotations with separate write buffers of depth 6 proves to provide the highest performance. But the gain of going from depth 5 to depth 6 is marginal.

On some benchmarks, single rotation outperforms multiple rotations, albeit not by much. For the 3GPP-LTE benchmarks however, multiple rotations with separate queues is by far the preferable implementation. The reason is that our implementations of these benchmarks involve both small and large power-of-two array access strides, up to

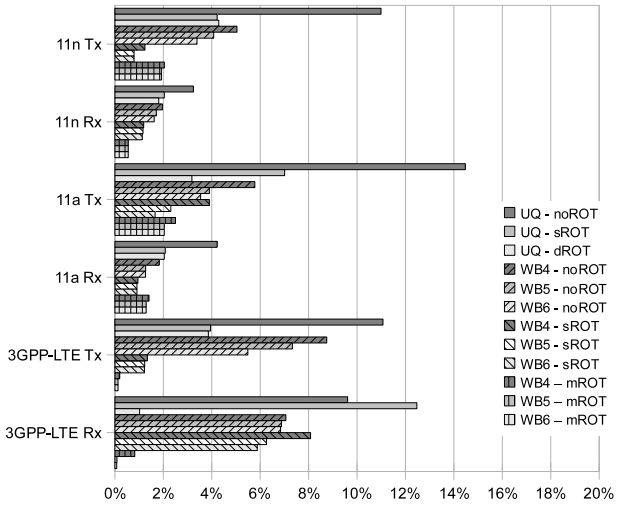

Figure 10. Percentage of the total cycle count that are stall cycles.

4096. The only way to handle all of them efficiently in hardware is to use multiple rotations. One could argue that in this case rewriting the software to avoid those large strides might also get rid of many conflicts. While that is true, the whole idea of software-defined radio is to enable shorter times to market. As such, the burden of optimizing code for a specific memory organization should not be put on the programmer. This is particularly so if the hardware solution involves little overhead in terms of energy or area as shown in the next sections.

5.4. Energy Results

Figure 11 shows the fraction of the total energy that is consumed in the 12 memory organizations we evaluated. These fractions include the power consumption in the queues and buffers. It can be seen that the differences in consumption between different memory organizations are small.

Between 15% and 20% of the 18% to 33% power consumed in the memory organization is spent in the queues and buffers. This means that our proposed conflict resolution schemes consume between 4% and 7% of the total power budget of an ADRES core including memories, depending on the benchmark and the specific scheme chosen. Considering the huge performance penalties as shown in Figure 2 for designs without conflict resolution, this energy overhead can be considered very low.

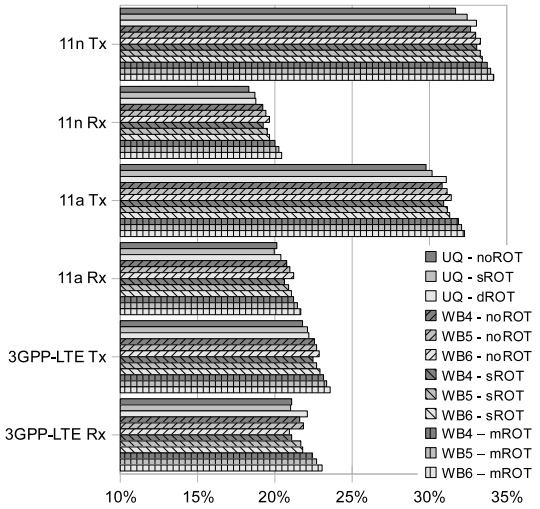

Figure 11. Percentage of energy consumption of the total core that is spent in the data memory subsystem.

It is no surprise that the benchmarks with the highest memory pressure consume most energy in the memory subsystem. It is also for those benchmarks that the highest gains in performance were obtained. As such, the increases in relative energy consumption as seen in Figure 11 must be interpreted correctly: rather than resulting from increased energy consumption in the memory subsystem, they result from reduced energy in the whole core as a result of the reduced number of stalls. This becomes obvious in the energy/performance plots of Figure 12. Every point in these plots shows, for positive numbers, the gains in total core energy consumption and the gains in execution time. In these charts, the scale of the energy gains is much smaller than the scale of the performance gains. The reason is that, because of clock-gating, stall cycles consume much less energy than active duty cycles. So only a limited amount of energy can be saved by eliminating stall cycles. In fact, these savings are so small that they are offset by the increased energy consumption in the queues.

5.5. Area Results

The basic unified queues without rotation take about 5.8% of the total core area on the SDR platform described in Section 5.1. This 5.8% includes all the hardware as discussed in Sections 4.1 to 4.4, except for the pipeline stages of the load/store units, and except for the memory banks themselves.

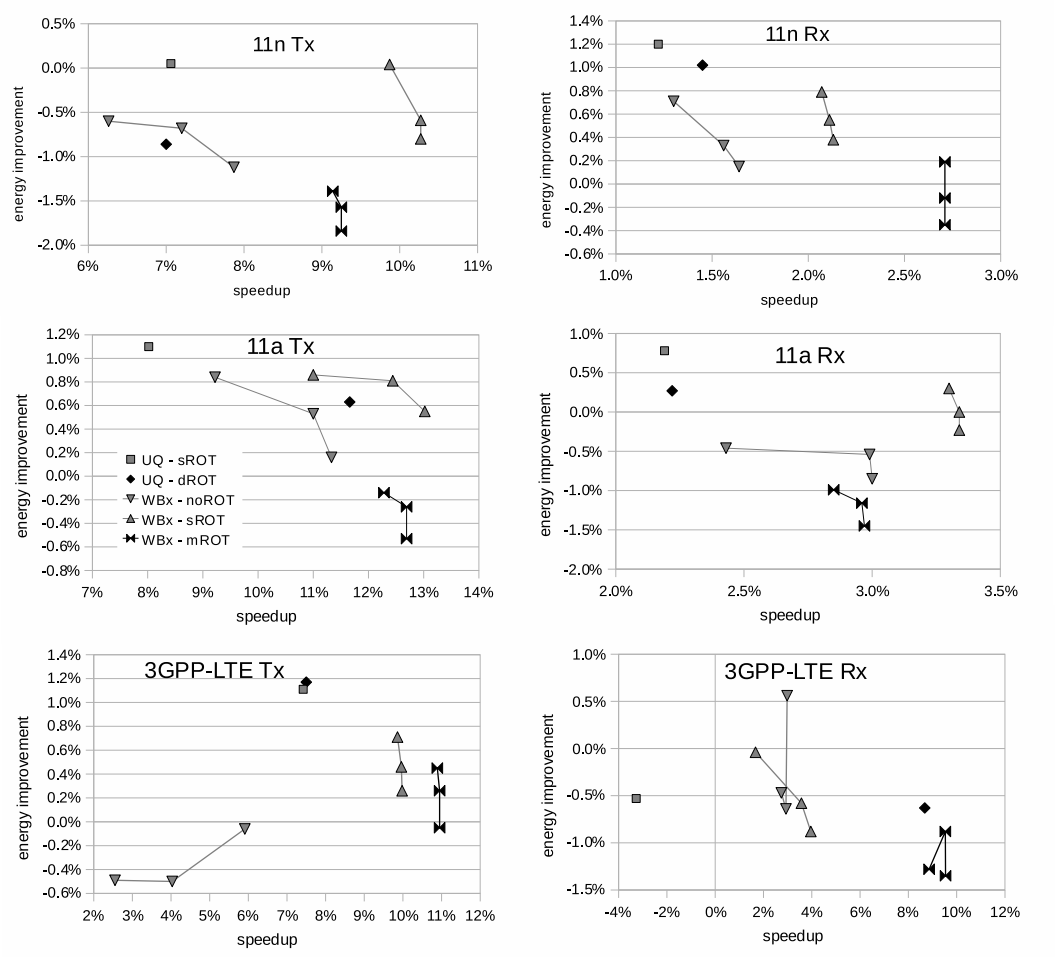

Figure 12. Energy savings and performance (execution cycles) improvements over the unified queues with non rotated banks for several standards, and for several organizations. The WBx lines connect WB4, WB5, WB6 from left to right.

Adding rotation does not increase the size of the memory interface significantly. But adding separate write buffers does increase it. Separate write buffers of size 4 makes the conflict resolution hardware grow 22% in size. For buffers of sizes 5 and 6, this becomes 33% and 45% resp. So with separate write buffers of sizes 4, 5 and 6, the conflict resolution hardware consumes 7.1%, 7.7% and 8.5% resp. of the total core area. All in all, the proposed conflict resolution mechanisms thus occupy little area.

5.6. Summary

Except for 11n and 11a Rx, significant speedups are achieved, up to over 13% for 11a Tx. This proves that the (combined) extensions of rotated banks and separate write-buffers are very useful. When no separate write buffers are used, multiple rotations is either significantly better or almost equally good as single rotation with respect to performance, while it never consumes more than 1% additional energy. Without rotation, separate write buffers prove to bring some gains in performance, but not nearly as much as with single rotation. Finally, it is clear from these figures that the combination of multiple rotations with separate write buffers is only useful for the advanced 3GPP-LTE benchmarks.

We can summarize these results as follows. Given that a SDR platform will run at least a couple of standards, which design is favorable depends on the precise standards that need to run and on whether energy or performance has higher priority. A choice can be made between UQ-mROT, WB4-sROT, WB5-sROT, or WB6-sROT or WB5-mROT. Alternatively, a more flexible design that supports switching between these modes could also be considered.

6. Related Work

Many reconfigurable architectures feature multiple independent memory banks/blocks to achieve high data bandwidth. But exploiting them automatically in a compiler is not a fully solved problem. RAW features an independent memory block in each tile (Taylor et al., 2002) for which Barua (Barua, 2000) developed a method called modulo unrolling to disambiguate and assign data to different banks. However, his technique can only handle array references whose index expression are affine functions of loop induction variables. MorphoSys has a very wide frame buffer (256-bit) between the main memory and a reconfigurable array (Singh et al., 2000). Its efficent use of such a wide memory depends by and large on manual data placement and operation scheduling. Both SiliconHive (Hive, 2007) and PACT (PACT, 2006) feature distributed memory blocks without crossbar. We did not find any public information in how their compilers map data onto such distributed memory blocks.

Zhuang *et al* developed a technique to treat the bank assignment problem as a graph coloring problem (Zhuang et al., 2002). It reorders individual accesses and operations to increase memory parallelism. Byoungro So *et al* present a compiler-based approach to derive custom data layouts in multiple memory banks for array-based computations (So et al., 2004). (Delaluz et al., 2002) describes a technique that

interleaves data for SDRAMs. Other techniques like (Zhang et al., 2000) also propose interleaving in SDRAMs while preserving the the locality. Similar to Barua's work (Barua, 2000), all these software-only techniques are very restrictive on what code can be handled. Typically, they cannot handle pointer-rich and data-dependent code. Most of these techniques target higher levels of memories (SDRAMs), and therefore are obliged to maintain locality. Furthermore, these techniques ignore the costs that may be incurred due to increased addressing complexity. Nonetheless, these software techniques still can be combined with our memory queue approach if they are applicable.

More complex hardware schemes like (Chen and Postula, 2000) improve the memory bandwidth bottleneck. However, they do introduce substantial hardware overhead like extra address generator units (AGUs) and look-up tables (LUT) for ensuring optimal memory interleaving. Such overheads are not acceptable on low power platforms.

Many memory interleaving schemes (a.k.a. skewing schemes) have been proposed in the past, such as (Rau, 1991; Pitkänen et al., 2008; Tanskanen and R. Creutzburg, 2005), of which some are much more complex than our simple bank rotation schemes. Harper et al (Harper and Linebarger, 1991) proposed a dynamic rotation scheme, in which special instructions can set the specific form of rotation for each loop. Thus they avoid the need to find a single scheme that fits all loops and their access patterns or strides. Multiple combinations of different dynamic schemes, including XOR-based, linear and rotation-like address mapping schemes have been investigated. Harper(Harper, 1991) builds on existing work to study the extent to which they can provide low-latency conflict-free accesses for a number of important data access patterns. His low-latency requirement excluded the use of buffers or queues. Valero et al. (Valero et al., 1995) discussed how the reordering of stream in vector processors or out-of-order processors accesses can result in conflict-free accesses for a wide range of strides.

The main difference between those papers and this paper is that we have demonstrated that low-latency, conflict-free memory addressing schemes are not needed to get close-to-optimal performance for SDR inner modem baseband processing with high-ILP processors. Thus we have shown that, e.g., support for dynamic addressing schemes is not needed for SDR inner modem baseband processing. Instead, combining a fixed address rotation with local load/store-reordering and memory access queues that exploit longer latencies suffices. We know of no literature in which the interleaving schemes of scratch-pad memories has been combined with load/store-reordering and with memory access queues that exploit longer instruction latencies.

Load/store queues have been used in modern out-of-order microprocessors (Park et al., 2003; Sethumadhavan et al., 2007; Castro et al., 2005; Subramaniam and Loh, 2006). They are used to absorb burst requests in cache accesses and to maintain the order of memory operations in a superscalar machine without having to wait until stores are actually committed to memory. Rivers *et al* describes a multi-bank cache organization that exploits data access locality (Rivers et al., 1997), where load/store queues are also used to meet buffering and ordering requirements of the superscalar machine. Hur and Lin (Hur and Lin, 2007) target modern out-of-order processors with their adaptive history-based memory access scheduler. This scheduler takes global access history information into account to reschedule memory operations. Memory queues like those mentioned (Sethumadhavan et al., 2007) target ILP-rich architecture like TRIPS, at the expense of considerable hardware like content accessible memories (CAMs), Bloom filters, etc. Such complex techniques are used to honor memory dependences on their out-of-order processors. Additionally, they are not designed with predefined latencies as found in static schedules. Furthermore, the target applications of these load/store queues are general purpose computing, where very few properties of the application cannot be exploited at compile-time. Furthermore techniques like (Subramaniam and Loh, 2006) do not exhibit consistent behaviour as is required for real-time systems like SDRs.

7. Conclusions

SDR inner modem baseband processing applications that are executed on high-ILP processors such as wide VLIWs or CGRAs put high pressure on memory. This paper proposed bank conflict resolution schemes based on queues, buffers and bank assignment rotation to limit the number of stall cycles resulting from conflicting memory bank accesses. With low overhead in terms of energy and area, the proposed schemes enable the use of power-efficient single-ported memory banks, while limiting the number of stall cycles to below 2% for the 11a|n standards and below 0.13% for the demanding 3GPP-LTE standard. Thus the proposed hardware solution removes the burden of optimizing an application's data layout and access pattern strides from the programmer.

References

Baert, T., E. De Greef, E. Brockmeyer, P. Avasare, G. Vanmeerbeeck, and J.-Y. Mignolet: 2008, 'An Automatic Scratch Pad Memory Management Tool and MPEG-4 Encoder Case Study'. In: *Proc. of Design Automation Conference (DAC 2008)*. To appear.

Barua, R.: 2000, 'Maps: a compiler-managed memory system for software-exposed architectures'. Ph.D. thesis, Massachusetss Institute of Technology.

Bastoul, C., A. Cohen, S. Girbal, S. Sharma, and O. Temam: 2003, 'Putting polyhedral loop transformations to work'. In: *In Proc. Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*. pp. 23–30.

Bougard, B., B. De Sutter, S. Rabou, S. Dupont, O. Allam, D. Novo, R. Vandebriel, V. Derudder, M. Glassee, and L. Van Der Perre: 2008a, 'A Coarse-Grained Array Based Baseband Processor for 100MBPS+ Software Defined Radio'. In: *Proc. Of Design, Automation, and Test in Europe (DATE 2008)*.

Bougard, B., B. De Sutter, D. Verkest, L. Van der Perre, and R. Lauwereins: 2008b, 'A Coarse-Grained Array Accelerator for Software-Defined Radio Baseband Processing'. *IEEE Micro* **28**(4), 41–50.

Castro, F., D. Chaver, L. Pinuel, M. Prieto, F. Tirado, and M. Huang: 2-5 Oct. 2005, 'Load-store queue management: an energy-efficient design based on a state-filtering mechanism'. *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on* pp. 617–624.

Catthoor, F., K. Danckaert, C. Kulkarni, E. Brockmeyer, P. Kjeldsberg, T. Van Achteren, and T. Omnes: 2002, *Data access and storage management for embedded programmable processors*. Kluwer Acad. Publ.

Chen, S. and A. Postula: Feb 2000, 'Synthesis of custom interleaved memory systems'. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* **8**(1), 74–83.

Delaluz, V., M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, A. Sivasubramaniam, and I. Kolcu: 2002, 'Compiler-Directed Array Interleaving for Reducing Energy in Multi-Bank Memories'. In: *VLSI Design*. pp. 288–293.

Derudder, V., B. Bougard, A. Couvreur, A. Dewilde, S. Dupont, A. Folens, L. Hollevoet, F. Naessens, D. Novo, P. Raghavan, T. Schuster, K. Stinkens, J.-W. Weijers, and L. Van der Perre: 2009, 'A 200Mbps+ 2.14nJ/b digital baseband multi processor system-on-chip for SDRs'. In *Proc of VLSI Symposum*.

De Sutter, B., P. Coene, T. Vander Aa, and B. Mei, B.: 2008, 'Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays'. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. pp. 151–160.

Friedman, S., A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, S.: 2009, 'SPR: an architecture-adaptive CGRA mapping tool'. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*. pp. 191–200.

Harper, D.: 1991, 'Block, Multistride Vectorm and FFT Accesses in Parallel Memory Systems'. *IEEE Trans. on Parallel and Distributed Systems* **2**(1), 43–51.

Harper, D. and D. Linebarger: 1991, 'Conflict-Free Vector Access Using a Dynamic Storage Scheme'. *IEEE Trans. on Comp.* **40**(3), 276–283.

Hive, 'HiveFlex CSP2000 series, Programmable OFDM Communication Signal Processor'. http://www.siliconhive.com.

Hur, I. and C. Lin: 2007, 'Memory scheduling for modern microprocessors'. *ACM Trans. Comput. Syst.* **25**(4), 10.

Kandemir, M., J. Ramanujam, and A. Choudhary: 1999, 'Improving cache locality by a combination of loop and data transformations'. *IEEE Trans. on Computers* **48**(2), 159–167.

Lam, M. S.: 1988, 'Software pipelining: an effecive scheduling technique for VLIW machines'. In: *Proc. PLDI*. pp. 318–327.

Mahlke, S., D. Lin, C. W.Y., R. Hank, and R. Bringmann: 1992, 'Effective compiler support for predicated execution using the hyperblock'. In: *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*. pp. 45–54.

Mei, B., S. Vernalde, D. Verkest, and R. Lauwereins: 2004, 'Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Arcchitecture: a Case Study'. In: *Proc. of Design, Automation and Test in Europe (DATE 2004)*. pp. 1224–1229.

Mei, B., S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins: 2003, 'Exploiting Loop-Level Parallelism for Coarse-Grained Reconfigurable Architecture Using Modulo Scheduling'. *IEE Proceedings: Computer and Digital Techniques* **150**(5).

Novo, D., T. Schuster, B. Bougard, A. Lambrechts, L. Van der Perre, and F. Catthoor: 2008, 'Energy-Performance Exploration of a CGA-Based SDR Processor.'. *Journal of Signal Processing Systems*.

Oh, T., B. Egger, H. Park, and S. Mahlke: 2009, 'Recurrence Cycle Aware Modulo Scheduling for Coarse-Grained Reconfiguralbe Architectures'. In *Proceedings of the 2009 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. pp. 21–30.

PACT: 2006, 'PACT XPP Technologies'. http://www.pactcorp.com.

Park, I., C. L. Ooi, and T. N. Vijaykumar: 2003, 'Reducing Design Complexity of the Load/Store Queue'. In: *Proc. of the 36th International Symposium on Microarchitecture (MICRO-36)*.

Park, H., K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-S. Kim: 2008, 'Edge-centric modulo scheduling for coarse-grained reconfigurable architectures'. In: *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. pp. 166–176.

Pitkänen, T., J. Tanskanen, R. Mäkinen, and J. Takala: 2008, 'Parallel Memory Architecture for Application-Specific Instruction-Set Processors'. *Journal of Signal Processing Systems*.

Rau, B. R.: 1991, 'Pseudo-randomly interleaved memory'. In: *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*. pp. 74–83.

Rau, B. R.: 1995, 'Iterative Modulo Scheduling'. Technical report, Hewlett-Packard Lab: HPL-94-115.

Rivers, J. A., G. S. Tyson, E. S.Davidson, and T. M. Austin: 1997, 'On High-Bandwidth Data Cache Design for Multi-Issue Processors'. In: *Proc. of the 30th International Symposium on Microarchitecture (MICRO-30)*.

Sethumadhavan, S., F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler: 2007, 'Late-binding: enabling unordered load-store queues.'. In: D. M. Tullsen and B. Calder (eds.): *ISCA*. pp. 347–357, ACM.

Singh, H., M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho: 2000, 'MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications'. *IEEE Trans. on Computers* **49**(5), 465–481.

So, B., M. W.Hall, and H. E. Ziegler: 2004, 'Custom Data Layout for Memory Parallelism'. In: *Proc. of International Symposium on Code Generation and Optimization (CGO)*.

Subramaniam, S. and G. H. Loh: 2006, 'Fire-and-Forget: Load/Store Scheduling with No Store Queue at All'. In: *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. pp. 273–284.

Tanskanen, J. and a. J. N. R. Creutzburg: 2005, 'On Design of Parallel Memory Access Schemes for Video Coding'. *Journal of VLSI Signal Processing Systems* **40**, 215–237.

Taylor, M., J. Kim, J. Miller, D. Wentzla, F. Ghodrat, B. Greenwald, H. Ho, m Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Frank, S. Amarasinghe, and A. Agarwal: 2002, 'The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs'. *IEEE Micro* **22**(2), 25–35.

Valero, L. , T. Lang, M. Peiron, and E. Ayguadé: 1995, 'Conflict-free access for streams in multimodule memories'. *IEEE Trans. on Computers* **44**(5), 634–646.

van Berkel, K., F. Heinle, P. Meuwissen, K. Moerman, and M. Weiss: 2005, 'Vector Processing as an Enabler for Software-Defined Radio in Handheld Devices'. *EURASIP Journal on Applied Signal Processing* **2005**, 2613–2625.

Wehmeyer, L. and P. Marwedel: 2006, *Fast, Efficient, and Predictable Memory Accesses*. Springer.

Zhang, Z., Z. Zhu, and X. Zhang: 2000, 'A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality'. In: *International Symposium on Microarchitecture*. pp. 32–41.

Zhuang, X., S. Pande, and J. S. G. Jr.: 2002, 'A Framework for Parallelizing Load/Stores on Embedded Processors'. In: *Proc. of International Conference on Parallel Architectures and Compilation Techniques (PACT 2002)*.