This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2493518, IEEE Transactions on Computers

An Energy-Efficient Memory Unit for Clustered Microarchitectures

Stefan Bieschewski, Joan-Manuel Parcerisa,
and Antonio González, *Fellow, IEEE*

Abstract — Whereas clustered microarchitectures themselves have been extensively studied, the memory units for these clustered microarchitectures have received relatively little attention. This article discusses some of the inherent challenges of clustered memory units and shows how these can be overcome. Clustered memory pipelines work well with the late allocation of load/store queue entries and physically unordered queues. Yet this approach has characteristic problems such as queue overflows and allocation patterns that lead to deadlocks. We propose techniques to solve each of these problems and show that a distributed memory unit can offer significant energy savings and speedups over a centralized unit. For instance, compared to a centralized cache with a load/store queue of 64/24 entries, our four-cluster distributed memory unit with load/store queues of 16/8 entries each consumes 31% less energy and performs 4,7% better on SPECint and consumes 36% less energy and performs 7% better for SPECfp.

Index Terms—cache memories, microprocessors, parallel architectures, distributed architectures, clustered architectures.

1 Introduction

Despite the success of current symmetric chip multiprocessors (CMP) to exploit thread-level parallelism (TLP), because of Amdahl's law, the rates of performance improvement by TLP will start to decrease if single-thread performance does not improve accordingly. [1] This perspective favors asymmetric (or hybrid) multicore processors, which combine one or a few big cores with multiple smaller cores. A big core is better suited to execute sequential program sections where all cores—except one—are idle. In this situation, even small increments in sequential performance are magnified by a degree that justifies the use of techniques that would be deemed inefficient in the absence of CMP. [1]

Because the thermal and power budgets per chip are limited, increased energy efficiency and smart handling of wire delays translate directly into higher single-thread performance, especially for the big cores alluded above. Clustered (or partitioned) microarchitectures are a well-known architectural paradigm that tackles the wire delay problem, keeps the critical circuit complexity low, and is especially power-efficient. [2]

Although the design of clustered microarchitectures has been studied before, the partitioning of the memory unit has received little attention. [5][6][7][8] This paper proposes effective solutions to partition the memory pipeline and the first level data cache in the context of a "big core" four-clustered microarchitecture. Our distributed approach is more power-efficient than a centralized memory pipeline and our proposals provide significant speedups over a state-of-the-art baseline distributed memory unit.

2 Related Work

Traditionally, load/store queue entries are assigned to memory instructions in program order during the dispatch stage before they enter the out-of-order core. Only after the address was calculated and when the instruction enters the memory pipeline, is its queue entry occupied. The entry remains occupied until the instruction commits.

Reserving entries in the load/store queues of a distributed memory unit is more difficult because during the Dispatch stage the address is still unknown, and it is

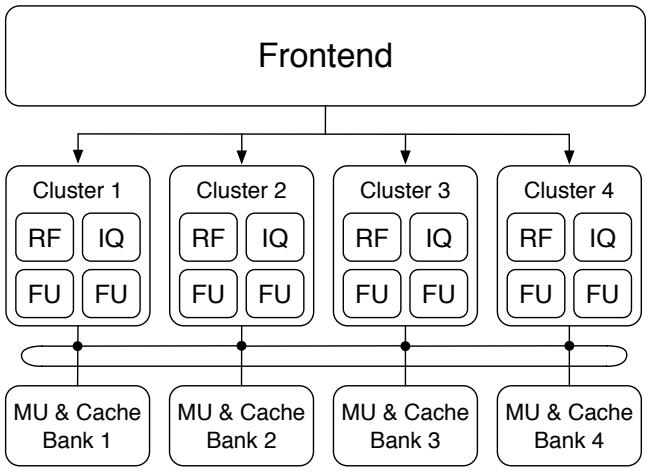

Fig. 1. Block diagram of the clustered microarchitecture. The frontend dispatches instructions to the backend, which consists of four clusters each with a register file (RF), issue queue (IQ), and functional units (FU). The clusters communicate through an interconnection network with a ring topology. Each cluster has an associated memory unit (MU) with one data cache bank.

- *Stefan Bieschewski is an independent consultant. E-mail: stefan@bieschewski.name.*
- *Joan-Manuel Parcerisa is with the Department of Computer Architecture, Universitat Politècnica de Catalunya, BarcelonaTech. E-mail: jmanel@ac.upc.edu*
- *Antonio González is with the Department of Computer Architecture, Universitat Politècnica de Catalunya, BarcelonaTech. E-mail: antonio@ac.upc.edu*

Submitted 11 June 2015.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2493518, IEEE Transactions on Computers

therefore not clear in which of the multiple queues an entry should be reserved. Yoaz et al. [5] propose to use bank prediction to reserve entries in two load queues. Zyuban and Kogge [2] extend this mechanism to clustered microarchitectures with up to eight banks. This extension exacerbates the number of mispredictions and the cost of replicating entries in multiple queues. Since bank mispredictions are discovered late in the pipeline, recovery requires either a pipeline flush and restart (similar to branch mispredictions) or the deallocation of its current queue entry and the allocation of a new entry in the correct queue. However, traditional load/store units do not support such migrations. Furthermore, there are no published bank predictors for many banks with an acceptable level of high–confidence mispredictions.

This paper presents a different approach to overcome this problem based on the *late allocation* of entries in physically unordered load/store queues [6][7][8]. It defers entry allocation until the address is calculated to avoid replication and the resulting need of large queues—with costly CAMs—or complex entry migration mechanisms that would be required otherwise. Physically unordered queues are challenging because they complicate the implementation of mechanisms such as disambiguation and store-to-load forwarding. This next section shows how a distributed memory unit with unordered queues can be implemented.

3 The Baseline Distributed Memory Unit

This section describes our baseline distributed memory unit. It combines several state-of-the-art techniques, such as late allocation of queue entries, no-hit bits, bank predictors, and store wait tables, which have not been proposed before in this combination for a distributed memory unit and which offer significant advantages over previous proposals. [2][7][13]

The clustered microarchitecture, which we use as starting point for our design, is described in greater detail elsewhere. [3] The superscalar architecture is divided into a centralized frontend and a distributed backend. The frontend consists of the Fetch, Decode, Cluster Steering, Rename, and Dispatch stages, the backend of the Issue, Register Read, Execute, Write–back, and Commit stages. Backend structures such as issue queues, register files, and functional units are all distributed over multiple clusters. In this section we describe an extension to this starting point that adds a distributed memory unit and a distributed first level data cache. Fig. 1 illustrates the main elements of the resulting architecture.

The cache banks are not interleaved word-wise like traditional cache banks, but cache-line-wise to avoid the replication of cache tags across clusters.

3.1 Inter-Cluster Networks and Instruction Steering

The distributed nature of the baseline architecture requires careful planning to minimize communications. Communications between clusters add latencies and consume significant amounts of energy.

Our baseline uses the Alpha ISA, where each load instruction has only one input register. The address calculation is always performed in the cluster that holds the input register. The calculated address determines which data cache bank the instruction will access.

The mapping of the output register determines into which register file the result will be written. To avoid another inter-cluster communication, the output register should be mapped preferably to the cluster with the corresponding cache bank. Because the data cache bank is unknown until the address calculation, the instruction steering mechanism employs a bank predictor. [5][10]

Store instructions have two input registers; one determines the memory address and the other the memory data. Internally, we split store instructions into store address and store data instructions. This allows the disambiguation logic to use the store address, even if the store data is still pending. The steering mechanism maps the two instructions to the same cluster.

3.2 Reservation and Release of Queue Entries

Instructions are inserted into the queue when they access the memory pipeline. Queue entries are allocated out-of-order and the queues are not physically ordered. The problem of allocating queue entries is very similar to allocating physical registers, which is a well-studied topic.

When a memory instruction commits, the ROB broadcasts a message to all clusters of the distributed memory unit. This message specifies how many load and how many store instructions are to be committed. The committed instructions are removed from the load/store queues. We discuss a more aggressive method to release load queue entries in section 7.

3.3 Store–to–Load Forwarding

Organizing the memory unit in banks guarantees that memory dependencies are always confined to a single cluster. To enable the forwarding of data from a store to a load instruction, load instructions search the store queue when they are executed.

If no match is detected the load obtains its data from the memory hierarchy.

If a single match is detected we compare the age of the load and store instructions. We enable store–to–load forwarding if the load is younger than the store. (i.e. if the load succeeds the store in the original instruction stream) To compare their age we simply subtract the nine-bit sequence numbers of the two instructions.

If more than a single match is detected we must identify the youngest of the matches that are older than the load. A physically unordered store queue requires a different approach than a physically ordered queue to handle this case. We adopt a scheme described by Webb, Keller, and Meyer [9]. They add a no–hit bit to all queue entries. Whenever a store instruction is inserted into the queue, its address is compared to the addresses of store instructions already present in the queue. If a store instruction in the queue matches, the no–hit bit of the older of the two instructions is set. At any point in time, out of several stores in the queue with matching addresses, all but the youngest will have the no–hit bit set. Instructions with the no–hit bit set are exempt from all future address

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2493518, IEEE Transactions on Computers

comparisons. As a result, load instructions that search the store queue never match more than a single store instruction and only a single circuit is required for age comparisons. At least two other more complex solutions have been published. [7] [11]

3.4 Handling Unresolved Stores

Load instructions execute speculatively and search only for store instructions present at execution time in the store queue. To verify that no dependencies were violated, store instructions search the load queue for younger loads (i.e. loads which succeed the store in the original instruction stream) with matching addresses as soon as they are inserted into the store queue. If such a load is identified the microarchitecture restores consistency with a pipeline flush. We provide only one comparator circuit for this test and use the no–hit bit for load queue entries in a similar manner as for the store queue described in section 3.3 above, i.e. for all the load instructions that match the address the no–hit bit is used to identify the youngest hit.

To reduce the number of memory dependency violations we utilize a store wait table similar to the Alpha EV6. Load instructions which are predicted to violate dependencies, are retained in the issue queue until the addresses of all older store instructions are known. Our experiments show that this arrangement consistently outperforms conservative issue policies as well as complex proposals, which involve address and dependency predictors. [13]

TABLE 1
MAIN SIMULATION PARAMETERS

Frontend (Monolithic, In-Order)	
width / depth / reorder buffer size	8 / 9 / 256
Backend (Clustered, Out-of-Order)	
number of clusters	4
ALU (per cluster)	
units / registers / issue queue size	2 / 56 / 16
FPU (per cluster)	
units / registers / issue queue size	1 / 56 / 16
Interconnection Network	
topology / latency per hop	ring / 1 cycle
Level 1 Data Cache	
Centralized cache (monolithic)	
size / associativity / line size	64KB / 2 / 32B
latency	3 cycles
ports	2 read/write
OR	
Clustered cache (per cluster)	
size /associativity / line size	16KB / 2 / 32B
latency	2 cycles
ports	1 read /1 write
Bank Predictor (monolithic)	
size	3.3KB
type	tournament
Other Memory Hierarchy	
Level 1 instr. cache (monolithic)	
size / associativity / line size	64KB / 2 / 32B
latency	1 cycle
Level 2 unified cache (monolithic)	
size / associativity / line size	2MB / 16 / 32B
latency	14 cycles
Main Memory	
latency random / latency burst	96 / 13 cycles

3.5 Multiprocessor Memory Consistency

The requirements for multiprocessor memory consistency vary with the ISA in question. In our experiments, we adopt the Alpha ISA and its memory consistency model. However, with the exception of section 6, everything described in this paper could be adapted to a stricter consistency model, e.g. sequential consistency.

The Alpha consistency model requires that we detect cases where two speculative load instructions access the same memory location in reverse order, where the younger load (i.e. the load that succeeds the other in the original instruction stream) accesses the location before the other load does.

To test for this condition each load instruction searches the load queue for instructions with the same address. As mentioned above in section 3.3 this search and the following age comparison are required to establish the no–hit bits in the load queue. This scheme is extended to detect the case where the load that is about to be inserted into the queue hits a load instruction that is younger (i.e. the load already in the queue succeeds the other load in the original instruction stream). When this case is detected, the pipeline is flushed to avoid a violation of the memory consistency model.

4 EXPERIMENTAL METHODOLOGY

To evaluate our microarchitectural proposals we use a simulator, which is based on the SimpleScalar toolset. We use the SPEC CPU2000 benchmarks to evaluate our architecture proposals. The benchmarks are compiled and optimized for the Alpha EV6 using the original toolchain from Digital. We simulate the first 100 million instructions after skipping the initialization phase of each benchmark. Table 1 summarizes the most important architectural parameters.

We use two first level data cache organizations in our experiments: A centralized organization with a single memory pipeline and a single data cache, as well as the distributed organization described in the last section. The centralized memory pipeline is connected to one of the clusters. The other three clusters have to use the interconnection network to access the memory pipeline.

We use CACTI 6.5 to calculate the latencies of the centralized cache and the distributed cache banks. While the latencies depend on the process technology in question, the ratio of the latencies for centralized and distributed caches remains approximately the same for technologies from 90 to 32nm, so that the results of the simulations are meaningful for various process technologies.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2493518, IEEE Transactions on Computers

5 Avoiding Load/Store queue overflows and deadlocks

In this section, we propose two extensions to the architecture we described in section 3. These extensions avoid queue overflows and deadlocks that would otherwise result in pipeline flushes.

Using late allocation in a distributed memory unit requires a trade-off for the queue size. In the worst case, when all load/store instructions access the same memory bank a queue has to hold all in-flight loads and stores. Oversizing the queues to cover this case is unattractive because load/store queue entries are expensive and the average utilization per entry would be low. The alternative are smaller queues and a mechanism to handle overflows. If an instruction overflows a load/store queue, it has to be re-executed. The architecture described so far lacks a mechanism to handle re-executions and recovers from an overflow with a pipeline flush.

In the same vein, a single memory pipeline cannot sustain the combined bandwidth of all address generation units. Because the memory pipelines cannot assert backpressure over the interconnection network, bursts of instructions to a single memory bank lead to overflows and pipeline flushes. Our first proposal the memory issue queue improves the handling of overflowing queues.

Another problem of the late, out-of-order allocation are deadlocks. When the calculation of an address is delayed so long that all queue entries in the corresponding cluster are already occupied by younger instructions a deadlock occurs. Because queue entries are deallocated in program order in the Commit stage, a younger instruction will deallocate its entry only after all older instructions committed. The architecture described so far uses a pipeline flush to recover from this condition. Our second proposal, Deadlock Aware Entry Allocation, avoids these deadlocks.

When we compare the extensions proposed in in this section we refer to the basic architecture outlined so far as naïve. For performance comparisons, we also include a configuration with a centralized memory pipeline and a centralized data cache. Because the load/store queue sizes of centralized and distributed memory pipelines are not directly comparable, we include only one centralized configuration with large load and store queues (64 and 24 entries respectively)—larger queues do not increase performance further. In section 7 we will compare centralized and distributed configurations in more detail.

5.1 Memory Issue Queue (MIQ)

In this section, we propose memory issue queues to decouple the memory pipelines from the address generation units. With memory issue queues of 16 entries, we can avoid load/store queue overflows and costly pipeline restarts almost completely. With a memory issue queue, the execution of memory instructions can be delayed until the required execution resources are available. This includes load/store queue entries as well as execution slots in the memory pipeline.

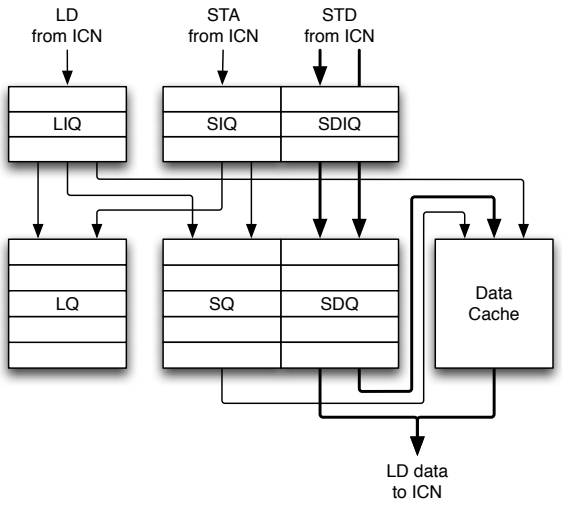

Fig. 7. Memory issue queue block diagram.

A larger load/store queue could reduce overflows too, but load/store queue entries contain CAMs, that make large queues slower and consume more energy.

Fig. 3 shows how the different memory issue queues fit into the design. The load issue queue (LIQ) receives load instructions from the interconnection network (ICN), buffers the instruction if necessary, and finally issues the instruction for execution in the memory unit. The store issue queue (SIQ) works in a similar way for store address instructions. The store data issue queue (SDIQ) is not an independent queue but essentially a buffer that holds store data instructions, while their corresponding store address instruction is waiting in the store issue queue.

Memory issue queues allow us to establish an oldest-first issue policy for memory instructions. This improves performance by approximately 3% over a simple FIFO. In addition, we adapt the issue policy to block instructions that are likely to cause pipeline flushes (e.g. deadlocks, memory dependency violations, etc.).

The presence of a memory issue queue implies that memory instructions will issue two times: first to calculate the address and then to access memory. These queues allow memory instructions to re-issue should an error occur during execution. Such errors include partial memory dependencies, store–load dependencies where the store data has not yet arrived, and others. Providing a re-issue mechanism allows the architecture to handle these cases more gracefully than e.g. with a pipeline flush.

Some microarchitectures allow instructions to re-issue directly from the load/store queue [14]. In these microarchitectures, the load/store queue additionally serves as memory issue queue, even though an individual queue entry only requires one of the two functions at any given time. By separating the queues we make better use of the complex logic.

Memory issue queues can be implemented using well-known techniques. Buyuktosunoglu et al. describe spatially unordered issue queues. They also detail mechanisms to select the oldest instruction in the queue. [12]

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2493518, IEEE Transactions on Computers

Instructions, which did not execute successfully, remain in the memory issue queue, but are marked as not ready. The queue includes a wakeup mechanism to promote these instructions again to the ready state [14] [15].

5.2 Deadlock Aware Entry Allocation (DAEA)

In this section, we propose a deadlock aware entry allocation scheme to avoid deadlock conditions. This technique reduces the number of pipeline flush events and allows us to use smaller load/store queues without sacrificing performance.

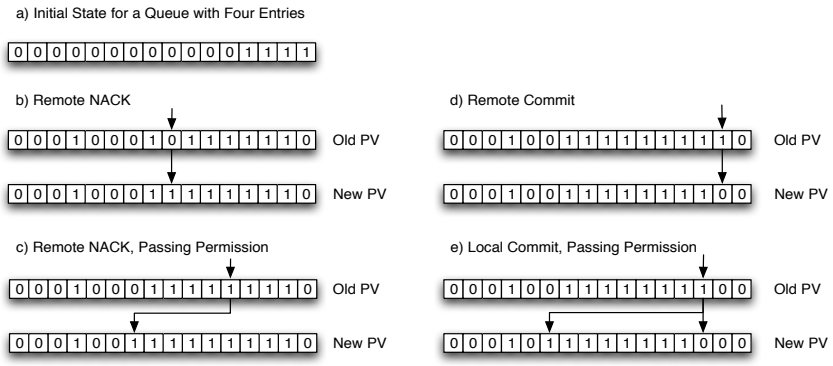

Fig. 4. Updates of the permission vector (PV) for the deadlock aware entry allocation.

The memory issue queue allows us to use an issue policy. Deadlock aware entry allocation is an issue policy that avoids deadlocks. The mechanism relies on information that is exchanged among the clusters over the interconnection network.

To avoid deadlocks completely an instruction must issue only if all older instructions are guaranteed to be able to issue too. This includes instructions whose addresses are not yet calculated. Once instructions calculate their addresses and hence their mapping to clusters becomes known, they inform all clusters of the mapping. To decide whether or not to issue an instruction, all instructions with unknown mappings and all instructions that map to the same cluster are taken into account. Only if the number of free entries exceeds the number of older instructions mapped to the same cluster plus the number of those with unknown mappings, is it safe to issue an instruction.

In the following discussion we will focus on the load queue, however the same principles apply to the store queue. The deadlock–aware entry allocation mechanism is built around a permission bit–vector (PV). Each memory issue queue has its own permission vector. This vector is indexed by the load sequence number and contains one bit for each dispatched, in-flight load instruction. An instruction is allowed to issue if its corresponding permission bit is set. The initial state after reset is a permission vector that contains all zeros except for the first N_{queue} positions, where N_{queue} refers to the size of the load queue. Fig. a) shows an example for a queue size of four entries. This vector allows the four oldest instructions in-flight to issue but denies issue to all other instructions.

As more instructions are executed, more information becomes known about the mapping of instructions to clusters. Whenever a load calculates its address, it is sent to the corresponding cluster and a NACK message containing the load sequence number is sent to all other clusters to inform them about the new mapping. Upon receiving the messages, the other clusters update their permission vectors in the following way: Fig. b) shows the example of a NACK message that is received for an instruction that had no prior permission to issue. In this case, its corresponding permission bit is set in the vector. Fig. c) shows the case of a NACK message for a load that did have prior issue permission. In this case, the permission is passed along to the oldest instruction without issue permission. This is accomplished by passing the permission bit to the left until the first zero-bit is encountered, somewhat similar to carry propagation.

At any time, the number of bits set to one in the permission vector is equal to the number of received NACK messages plus N_{queue}. The mechanism gives issue permission to the first N_{queue} instructions out of all instructions that did not send a NACK message. Notice that the instructions that did not send a NACK message are those which are known to map to the local cluster or whose mapping is not yet known. By choosing the first N_{queue} instructions out of these instructions, deadlocks are completely avoided.

To allow continuous operation the permission vector is organized as a circular buffer and the permission bits may pass from the leftmost to the rightmost bit. Upon commit of an instruction the corresponding permission bit is cleared in the remote clusters as shown in Fig. d). If the commit liberates an instruction in the local load queue, the next youngest load is given permission to issue, see Fig. e). Again, this is implemented by a carry-propagation-like mechanism that traverses the permission bits to the left until a zero bit is encountered.

To guarantee the correct behavior of this mechanism, the size of the permission vector must at least be equal to the maximum number of load instructions in-flight in the out-of-order core *plus* N_{queue} bits. These N_{queue} additional bits are necessary to handle extreme cases, for example, when a single cluster receives NACKs for all in-flight loads.

Clusters calculate addresses in parallel, and multiple NACK messages may be generated in a single cycle. In our experiments, we limit the addresses generation rate to one load and one store address per cluster per cycle. This matches the cache bandwidth of one load and one store per cluster.

5.3 Evaluation

Fig. 5 shows that the memory issue queue and the deadlock aware entry allocation reduce overflow and deadlock events. The memory issue queue reduces flush events significantly for all queue sizes. This reduction is also significant for large load queues, where overflows are primarily caused by the bandwidth mismatch between address generation units and memory pipelines. Small

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2493518, IEEE Transactions on Computers

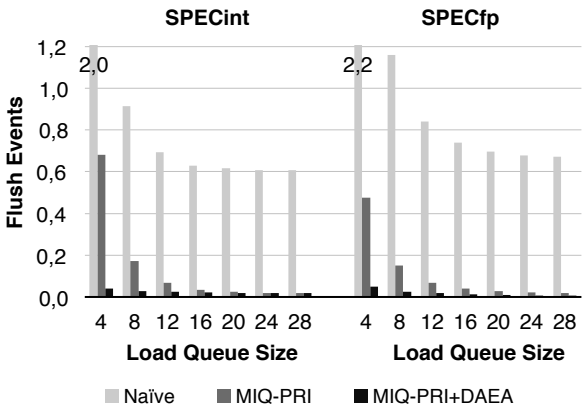

Fig. 5. Pipeline flush events caused by load queue overflows and load queue deadlocks per 100 committed instructions. The figure shows the load queue size of each of the four clusters for a naïve configuration and memory issue queues with oldest-first policy (MIQ-PRI) and additionally with deadlock aware entry allocation (MIQ-PRI+DAEA).

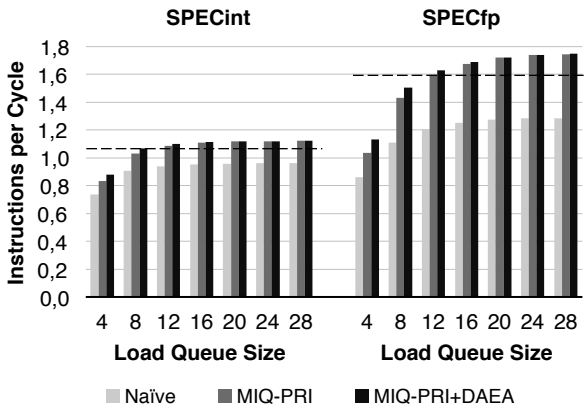

Fig. 6. Instructions per cycle for deadlock aware entry allocation. The figure shows the load queue size of each of the four clusters for a naïve configuration and memory issue queues with oldest-first policy (MIQ-PRI) and additionally with deadlock aware entry allocation (MIQ-PRI+DAEA). The dashed line shows the performance of a centralized memory pipeline with 64 load queue entries.

queues are especially prone to deadlocks, because a full queue is a precondition for a deadlock. The deadlock avoidance scheme eliminates all deadlocks; the few remaining flush events are caused by overflows.

Fig. 6 illustrates the performance impact of the memory issue queue and the deadlock avoidance scheme. The best configuration for a memory issue queue improves performance by 14.5% for integer and by 32.0% for floating-point benchmarks. The deadlock avoidance scheme significantly improves the performance for configurations with small queues. Because large queues suffer few deadlocks, they offer less potential for improvement. Small load queues, 16 entries for integer and 24 entries for floating-point, reach a performance plateau. Even smaller queues still obtain high performance. A load queue size of 12 entries can achieve integer performance within 1.9% of the largest load queues and 20 entries get within 1.3% of the best floating-point performance. The integer performance of a centralized load queue can be reached with only 8 queue entries. A queue with 12 entries surpasses the integer as well as the floating-point performance of a centralized queue.

6 Early Release of Load Queue Entries (ERLQ)

In this section we propose the early release of load queue entries. This technique allows smaller load queues without sacrificing performance.

To maximize the effective load queue size, instructions should occupy load queue entries no longer than required. Traditionally, instructions free their load queue entries when they commit. By freeing entries earlier, we can make better use of the load queue entries. This proposal can be applied not only to our distributed memory unit, but to centralized memory pipelines as well. However, this scheme is limited to ISAs with a relaxed memory consistency model. (Our other proposals are not restricted to a particular consistency model.)

The two common uses of the load queue are detection of data dependency misspeculations and enforcement of multiprocessor memory consistency.

Memory units, which use data dependency speculation and issue loads speculatively, usually employ the load queue to detect store-load-ordering violations, which occur when a load was reordered with respect to a store and both instructions access a common memory location. The key observation here is that store instructions search only for younger load instructions. When a load address has been compared to all older stores, its queue entry is no longer needed to detect store–load dependency violations.

Shared memory multiprocessor systems require further verifications of the memory ordering. Which verifications are required depends on the memory consistency model. For some ISAs like Alpha, PowerPC, ARMv7, SPARC RMO, and Itanium correctness can be guaranteed by asserting that load instructions from the same core do not access a memory location out of order. This can be implemented by searching the load queue for younger instructions with matching addresses, whenever a load instruction is executed. The key observation here is that load instructions search only for younger load instructions. When a load address has been compared to all older loads, its queue entry is no longer used to detect coherency violations.

If we combine the two observations, we arrive at a condition that allows us to release load queue entries before the commit stage without impacting the functionality of the load queue. In the case of a distributed memory unit, we must also consider instructions that are mapped to other pipelines. The condition to release a load queue entry is modified as follows: All load and store instructions which are older than the load in question must already have searched the load queue or are known to execute in other pipelines.

The mechanism for the early release of load queue entries keeps track of instructions, which execute locally as well as remotely. To keep track of remote memory in-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2493518, IEEE Transactions on Computers

structions it uses the same NACK messages that were used in the deadlock aware entry allocation, section 5.2 above. This information is aggregated in a bit-vector and used to decide if an instruction qualifies to be released early from the load queue.

Fig. shows the impact of the early release of load queue entries on performance. For small queue sizes, there is a significant benefit of early release and the performance plateau can be reached with smaller queues. In the case of integer benchmarks, a configuration with only 6 queue entries achieves performance within 3.8% of the largest configuration (with 8 entries within 2.2%). The floating-point benchmarks show even greater improvements in IPC. A configuration with 16 queue entries achieves floating-point performance within 2.3% of the largest configuration. The performance of a centralized load queue can be reached with only six entries for integer and eight entries for floating point benchmarks.

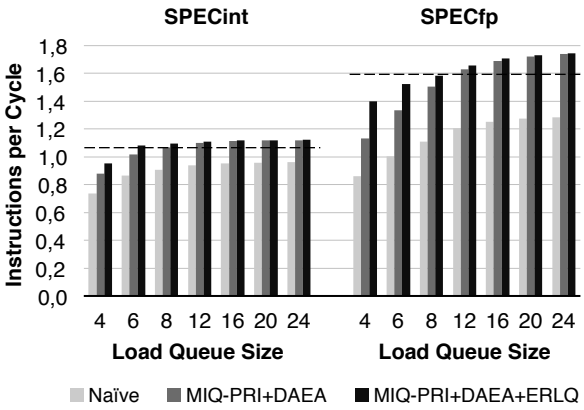

Fig. 7. Instructions per cycle for early release of load queue entries. The figure shows the load queue size of each of the four clusters for a naïve configuration and memory issue queues with oldest-first policy (MIQ-PRI). Notice that compared to the previous graphics we add the size 6 and omit the size 28. The dashed line shows the performance of a centralized memory pipeline with 64 load queue entries.

7 Performance and Energy Efficiency

To make the case for a distributed memory unit we conclude the paper with an estimation of performance and energy of our proposal compared to a centralized memory unit.

To estimate the energy of the memory unit we estimate the energy cost of different microarchitectural events and sum the quantities over the course of the simulation. We account for dynamic energy related to the memory unit including the data caches, the disambiguation logic, the bank predictors, and the inter-cluster communication. The individual energy costs are calculated using CACTI 6.5. For the sake of simplicity, we do not take into account memory issue queues, the translation look–aside buffer, cache misses, or external snoop events.

The centralized configuration uses a cache of 64 Kbytes size, two read/write ports, and 3-cycle access latency. The distributed configuration uses four caches of 16 Kbytes size each, one read port and one write port each, and 2-cycle access time. It also has a bank predictor, which is used to minimize communication between clusters. To calculate the length of the interconnects between two neighboring clusters, we roughly estimate the chip area of each clustered backend. We studied illustrated die plots of three out-of-order microarchitectures and measured the chip area of the blocks that correspond roughly to a cluster of the backend. These blocks account for two integer units, a single floating-point unit, the memory unit, the register file and the corresponding issue queues. By averaging the three estimates, we arrive at an approximate area of 1,600 million square lambda per cluster. We set the interconnection length of our model to the square root of the estimated cluster area. Interconnects are 78 bits wide, 64 bits of address or data and 14 bits of additional information such as instruction type, destination cluster, sequence number, access size, etc. We calculate the energy for the interconnects with the model from CACTI 6.5, the same model we also use for the data cache.

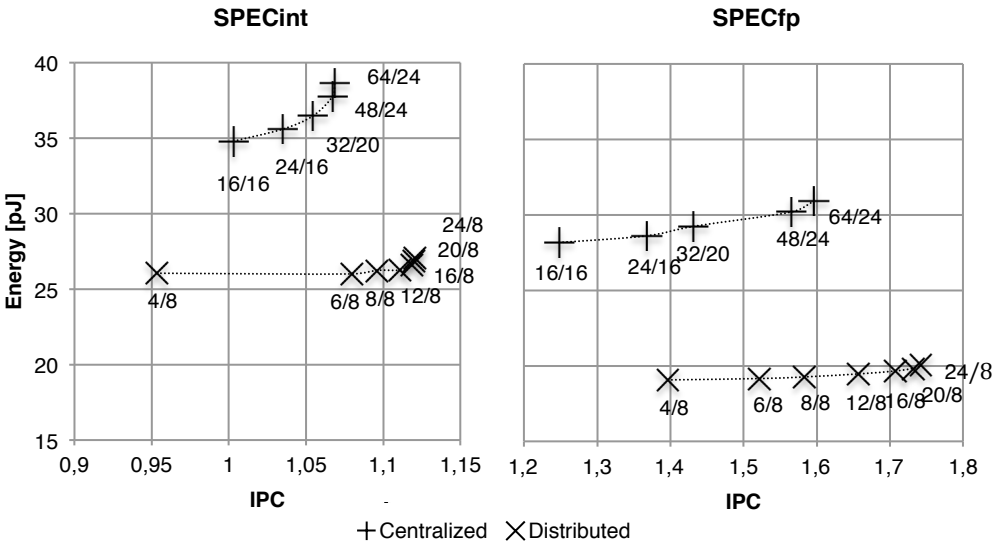

Fig. 8. Energy vs. Performance for centralized and distributed configurations. Only the energy usage of the memory unit and related inter-cluster communications are included. The numbers near each data point indicate the sizes of the load/store queues. In case of the distributed configuration the sizes refer to the queues in each cluster (of the four). The graph shows the results for a 32nm process, the scale on the left indicates picojoule per committed instruction. Graphs for 90nm, 65nm, and 45nm processes are very similar to the above and are not shown.

Fig. shows the results of this estimation. For SPECint as well as SPECfp the distributed configurations achieve higher performance and at the same time consume significantly less energy. As the sizes of load and store queues are increased, at some point only energy increases and performance stagnates. This can be observed in the integer part of the figure. The floating-point benchmarks can make use of larger

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2015.2493518, IEEE Transactions on Computers

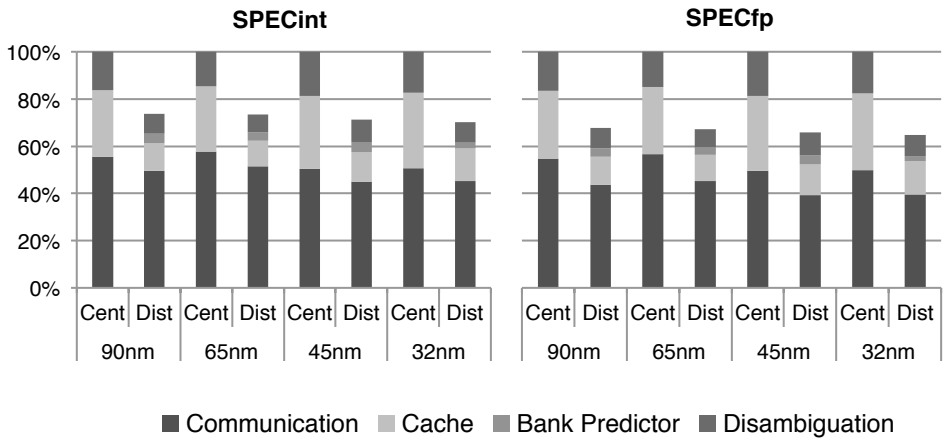

Fig. 9. Dynamic energy by component for centralized and distributed configurations. The centralized configuration shown includes a load queue of 64 entries and a store queue of 24 entries. The distributed configuration shown includes a load queue of 24 entries and a store queue of 8 entries per cluster. The data is normalized for the centralized configuration and each process technology.

queues and the effect is not as visible because we did not include huge queue sizes to illustrate this point.

The increased queue size affects primarily the cost of disambiguation. Since the centralized configuration contains larger queues it is affected more by increases in the size and the slope of the curve is higher than for the distributed configuration.

To understand these results better we break down the energy consumption of two configurations. We choose a centralized configuration with a load/store queue size of 64/24 entries and a distributed configuration with a load/store queue size of 24/8 for each cluster. Fig. shows the energy components and the trend for shrinking process technologies. The relative advantage of the distributed configuration increases slightly with smaller process technologies while the distribution of energy usage between the components remains stable. Not included in the figures is the increase in latency in smaller process technologies of the centralized cache relative to the distributed cache. For 90nm the centralized cache is 1.2 times slower than the distributed cache, for 32nm it is 1.7 times slower.

Comparing the energy usage of the different components, we observe that the distributed configuration uses less energy for each single component (except of the bank predictor). Communication is the component with the largest energy consumption. Even if we include the bank predictor into the communication cost, the distributed configurations use less energy for communication. The energy savings are most pronounced for caches and disambiguation. This is the result of using smaller structures.

8 Conclusion

This paper proposes several techniques to design a partitioned memory unit for a "big core" clustered microarchitecture, and to reduce the complexity of the disambiguation logic and the first level data cache. This approach provides significant energy savings and improved performance, compared to a centralized memory unit.

Our results show that the combination of our techniques can provide substantial energy savings and speedups to a four-cluster distributed memory unit, over a centralized approach. For instance, compared to a centralized approach with a load/store queue of 64/24 en-tries, partitioned memory units with load/store queues of 16/8 entries consume 31% less energy and perform 4,7% better on SPECint (36% and 7% for SPECfp). Even with load/store queues as small as 12/8 entries, the distributed memory unit provides significant energy savings and speedups over centralized configurations of any queue size.

References

[1] M. D. Hill, M. R. Marty, "Amdahl's Law in the Multicore Era," IEEE Computer July 2008, pp. 33–38.

[2] V.V. Zyuban and P.M. Kogge, "Inherently Lower–Power High–Performance Superscalar Architectures," 2001, HPCA–10, pp. 268–285.

[3] J.M. Parcerisa, "Design of Clustered Superscalar Microarchitectures," UPC, DAC, Ph.D. Thesis, 2004.

[4] S. Bieschewski, "Design of a Distributed Memory Unit for Clustered Microarchitectures", UPC, DAC, Ph.D. Thesis, 2014.

[5] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," 1999, ISCA–26, pp. 42–53.

[6] E. Torres, P. Ibanez, V. Vinals, and J. Llaberia, "Store buffer design in first–level multibanked data caches," 2005, ISCA–32, pp. 469–480.

[7] S. Sethumadhavan, F. Roesner, J.S. Emer, D. Burger, and S.W. Keckler, "Late–binding: enabling unordered load–store queues," 2007, ISCA–34, pp. 347–357.

[8] S. Bieschewski, J.-M. Parcerisa, A. González, "A Fully–Distributed First Level Memory Architecture," Technical Report UPC–DAC–RR–ARCO–2007–3, UPC, 2007.

[9] D. A. Webb, J.B. Keller, and D.R. Meyer, "Data cache having store queue bypass for out–of–order instruction execution and method for same," United States Patent no. 6360314, 2002.

[10] S. Bieschewski, J. Parcerisa, and A. Gonzalez, "Memory bank predictors," 2005, ICCD–23, pp. 666–668.

[11] E. Gunadi and M. Lipasti, "A position-insensitive finished store buffer," 2007, ICCD-25, pp. 105–112.

[12] A. Buyuktosunoglu, D.H. Albonesi, P. Bose, P.W. Cook, S.E. Schuster, "Tradeoffs in Power–Efficient Issue Queue Design," ISLPED, pp. 184–189.

[13] R. Balasubramonian, "Cluster prefetch: tolerating on-chip wire delays in clustered microarchitectures," 2004, ICS–18, pp. 326–335.

[14] J.M. Abramson, H. Akkary, A.F. Glew, G.J. Hinton, K.G. Konigsfeld, P.D. Madland, "Method and apparatus for performing load operations in a computer system", United States Patent no. 5694574, 1997.

[15] J. M. Tendler, S. Dodson, S. Fields, Hung Le, B. Sinharoy, "POWER4 System Microarchitecture," International Business Machines, Technical White Paper, 2001.