Design and Implementation of the TRIPS Primary Memory System

Simha Sethumadhavan, Robert McDonald, Rajagopalan Desikan *, Doug Burger and Stephen W. Keckler
Computer Architecture and Technology Laboratory
Department of Computer Sciences, The University of Texas at Austin
* Department of Electrical and Computer Engineering, The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

***Abstract*— In this paper, we describe the design and implementation of the primary memory system of the TRIPS processor. To match the aggressive execution bandwidth and support high levels of memory parallelism, the primary memory system is completely partitioned into four banks, can support up to 256 in-flight memory instructions, aggressive reordering of in-flight loads and stores, up to four loads and stores every cycle and up to 64 outstanding cache misses to sixteen different cache lines. The design was implemented using IBM 130nm ASIC technology and occupies 21% of the processor area. We describe in detail the microarchitecture of the memory system, detailed design of two of the most complex and interesting components – the LSQ and the MHU – and discuss the rationale behind some of the design decisions. Our design experience suggests that the complexity of the partitioned memory system is comparable to less aggressive centralized implementations.**

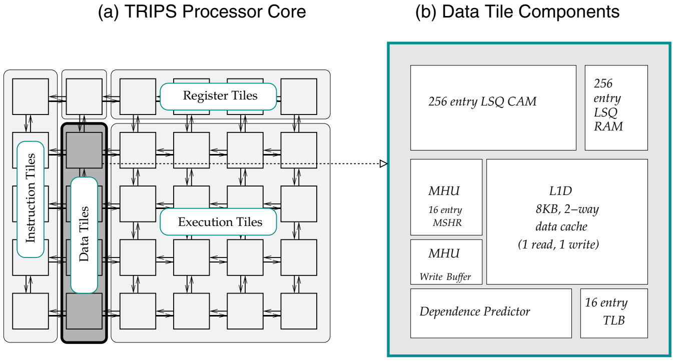
I. Introduction

Recent technology scaling trends are forcing more heavily partitioned microarchitectures [1], [2], [3]. In such designs, the memory system must also be partitioned to match the bandwidth and capacity requirements of the partitioned execution core. However, to our knowledge, no previous system has achieved a fully partitioned memory system that supports aggressive reordering of loads and stores while also supporting sequential memory semantics. In this paper, we describe the set of microarchitecture and design techniques that enable a fully partitioned level-one memory system, in which none of the necessary functions are centralized. We describe low-level design details of two of the most challenging portions of this partitioned design, the load/store queues and the miss handling unit.

Each TRIPS processor supports four thread contexts (similar to SMT), out-of-order execution, an instruction window of up to 1024 instructions, and up to 256 memory operations. To achieve high performance on this machine, the primary memory system must (1) be fully partitioned so as to provide memory bandwidth commensurate with the execution bandwidth, and (2) sustain a high degree of memory-level parallelism. For an aggressive execution core like TRIPS, a centralized L1 system cannot provide the necessary low latency, high bandwidth, or proximity.

To satisfy the above requirements, the TRIPS primary memory system implements the following advanced capabilities:

1) To support high bandwidth, the L1 caches are address-interleaved and partitioned across four cache banks thus supporting up to four loads and stores every cycle. In addition, the LSQ and MHU mechanisms are also completely partitioned. Previous implementations have used partitioning to a limited extent (like address interleaved caches [4]), but these architectures still used centralized structures such as the LSQ and MHU, limiting microarchitectural scalability.
2) To sustain high levels of memory parallelism, the TRIPS processor implements miss handling structures (“MSHRs”) that can support up to 64 load misses to up to sixteen cache lines. Most modern processors can support up to eight pending loads to four or fewer cache lines [5]. In addition, to support aggressive reordering of loads and stores, the TRIPS processor uses a *memory-side* dependence predictor. Typically, centralized processors feature fetch- or execution-side dependence predictors but these designs cannot be efficiently implemented in TRIPS because the fetch and execution portions of the microarchitecture are also partitioned.


(a) TRIPS Processor Core

(b) Data Tile Components

Fig. 1. Single core of the TRIPS SMT-CMP prototype and components of a single data tile

The primary memory system (level-1) of the TRIPS processor prototype is made up of four partitions, each called a Data Tile(DT). Each DT (Figure 1) is interleaved at a cache-line granularity and includes an 8KB, 2-way associative cache bank (L1D), a local copy of the load store queue (LSQ), a local copy of the data translation lookaside buffers (TLB), a store-load

1-4244-9707-X/06/$20.00 ©2006 IEEE

dependence predictor (DPR), and miss-handling unit (MHU). Each DT is connected to three different networks: the operand network delivers loads and stores from the execution units to the DTs, the L2 network is used to access the L2 cache banks on load and store misses and the status network connects all the DTs and is used to track stores arriving at the different DTs.

Using the structures and networks, each DT partition: (a) provides data for loads and stores, (b) performs address translation and protection with the DTLB, (c) handles cache misses with MHU, (d) tracks load and store dependences with LSQs, (e) performs load/store dependence prediction for aggressive load store issue with DPR, (f) detects when all store outputs for a TRIPS block have been produced, (g) writes stores to caches/memory when they become non-speculative, and (h) performs store merging on L1 store cache misses. Since each DT provides all necessary L1 capabilities for the addresses that are mapped to it, the design can be scaled in a complexity-effective manner by replicating the DTs.

The design, implementation and verification effort for the TRIPS partitioned memory system required a total 21 person months and used an IBM ASIC flow at 130nm for the design. A group of 4 DTs occupies 21% of the processor core area. Our design experience suggests that complexity is not a barrier to implementing partitioned memory systems. Although new mechanisms are required for partitioned systems, these mechanisms are simple, and the time-to-design and verification complexity are comparable to a centralized implementation.

The rest of this paper is organized as follows: Sections II and III describe the memory system related aspects of TRIPS architecture and the DT microarchitecture respectively. The design and implementation of the the LSQ and MHU – the two of the most interesting components in the design where the most innovation was required – are described in Sections IV and V. We describe the area and timing aspects of the design in Section VI and conclude with discussion of scalability and future work in Section VII.

II. Architecture Overview

The TRIPS system supports most features found in commercial architectures including different size loads/stores (8, 16, 32 and 64-bit), different types of memory attributes (mergeable/unmergeable, cacheable/uncacheable), synchronization instructions, and virtual memory. In addition, TRIPS ISA includes special support for providing sequential memory semantics and efficiently implementing a partitioned memory system.

Load/Store Ordering: To determine the correct memory order and thus track load store dependences in the partitioned primary memory system, the TRIPS processor uses specially encoded program order tags called Load Store IDs (or LSIDs). An LSID is a 5-bit field in a memory instruction. In most superscalar architectures, the program order (or "age") is determined dynamically in the fetch stage and hence the LSID is not included in the instruction. In a *completely* partitioned microarchitecture like TRIPS however, the fetch mechanism is also partitioned and there are multiple fetch points fetching independent instruction streams. In such cases, it is impossible to construct the total memory age order from the partial memory age orderings observed at different fetch points; hence LSIDs must be encoded as part of the instruction.

Consistency model: TRIPS implements a weak consistency model (similar to Power 4 [6]) that enforces load/store dependences but relaxes all execution orders and requires the programmer to insert memory barriers to realize more strict consistency. Direct hardware support of more traditional and programmer friendly consistency models on TRIPS (like in other aggressive out-of-order processors) can negatively impact performance. Consider, for instance, total store ordering (TSO). Among other requirements, TSO requires that stores to different memory addresses commit in program order. To support TSO on TRIPS, a store must be constrained to update memory only after the previous store has committed. This requirement would restrict store commits to one per cycle whereas a weaker model can enable simultaneous store commits from all four DT partitions. The slower deallocation of stores eventually leads to slower deallocation of other processor resources resulting in performance losses.

Block Atomic Execution: TRIPS implements a block atomic execution model in which a TRIPS block [2] can update architectural state only when all of its memory and register outputs have been generated. To support block atomic execution, each TRIPS block encodes the number of memory outputs for a block, and the LSIDs of all outputs in the store-mask bit vector, in the block header. This information is broadcast to all the DTs when a new block is fetched.

III. DT Microarchitecture Overview

The load and store instructions can be mapped onto any of the sixteen execution units on the TRIPS processor (Figure 1a.) The memory instructions issue from the execution units when all their inputs are available, and are then delivered to the DT through the operand network. This section provides an overview of the basic steps involved in load and store processing in the DT and the pipelines that implement the load/store processing steps. We also describe *store tracking*, the only additional microarchitecture mechanism required for implementing the TRIPS partitioned memory system. We conclude this section with a discussion of the rationale behind the memory side dependence prediction.

A. Load Processing

The pipeline diagram in Figure 2 illustrates the different stages of load processing. Every incoming load accesses (a) the TLB to perform address translation and check the protection attributes, (b) the dependence predictor (DPR) to check for possible store dependences, (c) the LSQ to identify older matching uncommitted stores, and (d) the cache tags to check for cache hits. Based on the responses (hit/miss) from the four units, the control logic decides on the course of action for that load. Table I summarizes the possible load execution scenarios in the DT.

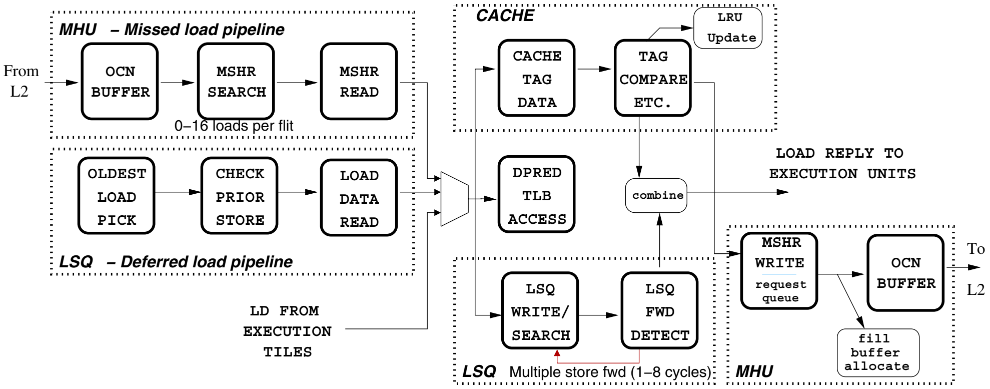


Fig. 2. The DT Load Pipelines

TLB	DPR	Cache	LSQ	Response
Miss	X	X	X	Report TLB Exception
Hit	Hit	X	X	Defer load until all prior stores are received
Hit	Miss	Hit	Miss	Forward data from cache
Hit	Miss	Miss	X	Forward data from L2 cache, issue cache fill request
Hit	Miss	Hit	Hit	Forward data from LSQ and cache

TABLE I
LOAD EXECUTION SCENARIOS. X REPRESENTS "DON'T CARE" STATE.

Load hit: When the load hits in the cache, and only in the cache, the load reply can be generated in two cycles. This best-case latency is likely to be the common case for most loads. When a load hits both in the cache and the LSQ, the load return value is formed by composing the values obtained from the LSQ and cache. First, the load picks up any matching stores bytes from the LSQ and then reads the remaining bytes from the cache. This operation can take multiple cycles and is referred to as store forwarding. Section IV describes store forwarding in more detail.

DPR hit: A load may arrive at the DT before an earlier store on which it depends. Processing such a load right away will result in a dependence violation and a flush leading to performance losses. To avoid this performance loss, the TRIPS processor employs a simple dependence predictor that predicts whether the load processing should be deferred. If the DPR predicts a likely dependence the load waits in the LSQ until all prior stores have arrived. After the arrival of all older stores, the load is enabled from the LSQ, and allowed to access the cache and the LSQs to obtain the most recent updated store value. Figure 2 illustrates the stages involved involved in processing deferred loads.

Load miss: If the load misses in the cache, it is buffered in the MSHRs [7] and a read request is generated and sent to the L2. When the data is returned from the L2, the loads in the MSHRs are enabled and load processing resumes. Like deferred loads, missed loads also access the LSQ and cache to pick up the most recent store values. Figure 2 illustrates the stages involved in processing missed loads.

B. Store Processing

Store processing occurs in two phases. During the first phase, each incoming store is buffered in the LSQ and the other DTs are notified about the store's arrival. During this phase each store checks for dependence violations; if any younger loads to the same address as the store are in the queue, then a violation is reported to the control unit, which initiates recovery. The dependence predictor is also trained to prevent such violations in the future.

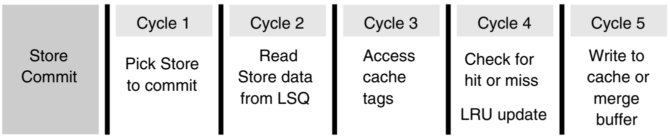


Fig. 3. The ST Commit Pipeline

When a block becomes non-speculative, the second phase of store processing begins as illustrated in Figure 3. In this phase the oldest store is removed from the LSQ, checked in the TLB, and the store value is written out to the cache/memory system. If the store hits in the cache, the corresponding cache line is marked as dirty. If the store misses in the cache, the store miss request is sent to the L2. We chose a write-back, write-noallocate policy to minimize the number of commit stalls.

C. Store Tracking

In the TRIPS execution model, a block can commit only after all of its store outputs have been generated. When a store arrives at any DT, the store arrival information is broadcasted to the other DTs through the Data Status Network (DSN). Each DT then increments a local counter that counts the number of stores that have arrived at the memory system. When all of the stores in a block have been received, the DT that received the last store sends a message to the control tile indicating that all memory outputs have been generated.

D. Memory-Side Dependence Processing

In the TRIPS implementation, the instruction window is partitioned across sixteen execution units (Figure 1) and the dependent loads and stores can be mapped on to any of the execution units. A naive extension of conventional dependence processing mechanisms [8] would hold back the load *in the execution unit* until the execution of the dependent store. This strategy can increse the load latency as explained below.

The latency of dependent loads can be broken down into four parts: (1) the latency for the load to detect that the dependent store has executed, (2) the latency for the load to be delivered from the execution unit to the DT, (3) the latency to access the DT, and (4) the latency to deliver the value from the DT to the target of the load. With execution side dependence processing, no overlap is possible between any of the latencies, because the loads are issued only after the dependent stores resolve and rest of the steps must be performed in order. However, memory side dependence processing allows the overlap of steps (1) and (2).

IV. LSQ Microarchitecture

The LSQ is critical for supporting aggressive memory ordering and is often considered to be one of the most complex structures in an out-of-order processor. This section describes the design of the LSQ in the context of the partitioned TRIPS microarchitecture.

The LSQ must support four major functions: (1) detect ordering violations between loads and stores to the same address, (2) forward values from uncommitted stores to loads to same address, (3) buffer and wake up deferred loads, and (4) buffer and commit store values to memory. All the above functions, except the third, are also required of the LSQs in conventional architectures. The third functionality is related to dependence prediction and is most efficiently implemented in the DT/LSQs for TRIPS. In this section, we describe the state in the LSQ and the store forwarding functionality of the LSQ. We omit the description of other functionality for space reasons.

A. LSQ state

The TRIPS microarchitecture allows up to eight blocks to be in-flight simultaneously and each block can have a maximum of 32 memory instructions; therefore, a maximum of 256 memory instructions in-flight. To accommodate the case when all the memory instructions in a block reach a DT partition, the LSQ is sized to hold 256 memory instructions. The logical and physical organization of the LSQ are illustrated in Figure 4.

B. LSQ Store Forwarding

The store-load forwarding operation is the most complicated operation in the LSQ because the load may match an arbitrary number of stores and can forward from up to eight distinct stores because of different sized LDs/STs. To handle this case, the LSQ scans all the matching stores starting from the most recent store, processing one matching store every cycle, either until the value every load byte has been obtained or until there are no matching stores to the unforwarded bytes.

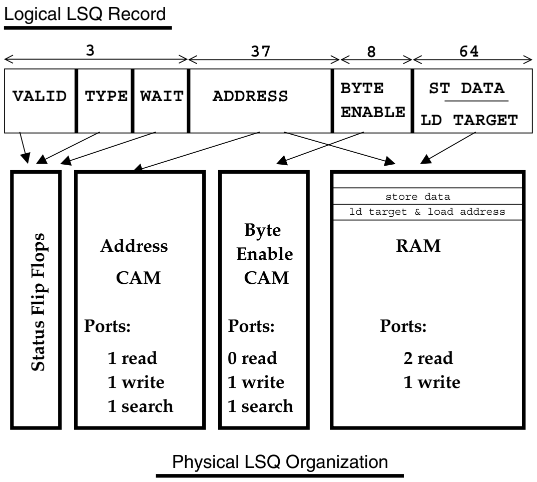


Fig. 4. Logical and Physical Organization of the LSQ

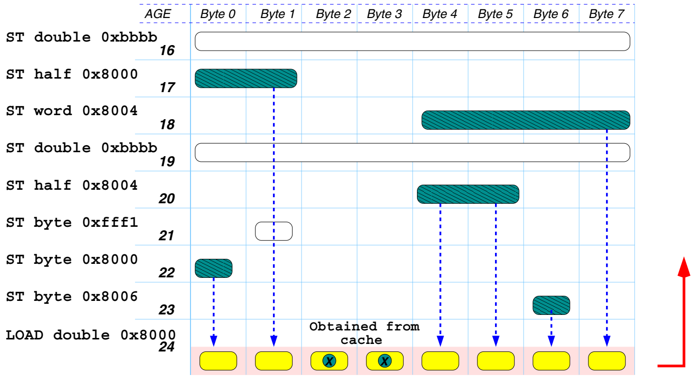


Fig. 5. Multiple stores forwarding to loads in the LSQ

Figure 5 illustrates an example of multiple forwarding where multiple stores of different sizes match to a 8-byte LD (Address 0x8000, Age 24). In the first cycle, the CAMs are associatively searched and the stores 23, 22, 20, 18 and 17 are identified as matching stores. These matches are scanned starting from the most recent store. In the first scan, byte 6 of the LD is retrieved from store 23 and load byte is marked as forwarded. In the next scan, a new associative search identifies the matching stores corresponding to the the remaining unforwarded bytes. The search now returns 22, 20, 18 and 17. Store 23 does not match because it produces byte 6 which was already forwarded to in cycle 1; at the end of the second scan, byte 0 from the store 22 is forwarded to the load. After the next three scans, bytes 4 and 5 are forwarded from 20, byte 7 is forwarded from 18 and byte 1 is forwarded from 17. At this point, there are no more matching stores to the remaining unforwarded bytes (bytes 2 and 3) and therefore LSQ forwarding is terminated. The remaining bytes are obtained from the cache. In this example, forwarding takes five cycles and during this period, the LSQ is completely stalled and cannot accept new loads or stores. Note that after the first associative search which searches both the address and the byte enable CAM and the rest of the scans search only the byte enable CAM which is eight-bits wide.

C. *Design Rationale*

Multiple forwarding in the LSQ: Many superscalar processors avoid the complex processing required for multiple forwarding by either simply flushing [9] or replaying [6] the load when multiple matching stores are detected in the LSQ. This strategy is not feasible in a block-atomic architecture like TRIPS because it can lead to deadlocks. If there are partial matches within a block, the load will not execute until the matching stores are drained, but the matching stores cannot be drained because the load and its dependents may have to execute to produce the block outputs.

Unified LSQ: Recently several processors have supported memory ordering using separate load and store buffers to increase the bandwidth and decrease the power per access. TRIPS, however, uses an unified LSQ because separate buffers will require more than 32 bits for encoding the memory instructions. This is because implementing separate LD and ST buffers requires that each memory instruction carry two different types of age tags; one tag encoding the global age and a second tag encoding the relative load/store ages. Alternatively, one can partition the TRIPS LSQ in LQ/SQ with the instruction encoding fixed at 32 bits. This strategy is disadvantageous because it restricts the number of memory instructions in the block and places hard restrictions on the number of loads and stores seperately in block (due to reduced number of bits available for both the tags).

Maximally sized LSQs: Although only one fourth of the total memory instructions are expected to reach any DT partition, the LSQ in each DT is maximally sized. There are two microarchitectural reasons motivating maximally sized LSQs:

1) Deadlock avoidance: If the LSQs are undersized then with speculative execution, younger memory instructions may arrive earlier than the older instructions and may take up all slots in the LSQ preventing the older instructions from completing and eventually stalling forward progress.
2) Design Simplicity: When we started the project it appeared to us that maximally sized conventional age-indexed LSQs would pose the least design risk because they were well-understood and straight-forward to implement.

In research conducted after the prototype implementation we have discovered complexity-effective methods to safely reduce the LSQ size without causing deadlocks [10]. These undersized LSQs are managed as a free-list and are as simple as the conventional LSQ.

V. Miss Handling Unit Microarchitecture

The Miss Handling Unit (MHU) plays a key role in sustaining high levels of memory parallelism by managing multiple, outstanding L1D load and store misses. The MHU sends L1D miss requests to the L2 cache via the On Chip Network (OCN) – the chip data transfer fabric – and receives read data (for load misses) and write acknowledgements (for store misses) from the L2 cache. While much of the TRIPS MHU functionality is typical of out-of-order processors, the use of on-chip networks imposes different correctness and performance requirements.

A. *MHU State*

Figure 6 illustrates the components of the MHU. The MHU in each DT contains (1) sixteen MSHRs that hold information on each of the missed loads, (2) four 64-byte fill buffers that hold cache lines returning from L2 (on the OCN) before it is written to the L1D cache, (3) a four entry FIFO load request queue(LRQ) that decouples fill buffer allocation from the load miss processing, (4) a 64-byte store merge buffer that can coalesce multiple stores to the same cache line before sending it to the L2, (5) a 64-byte store transmit buffer that is used as scratch storage for holding the coalesced writes while the OCN packets are being created and sent, and (6) a 64-byte spill buffer that holds one cache line worth of data and is used to temporarily buffer cache spills before sending them out on the OCN.

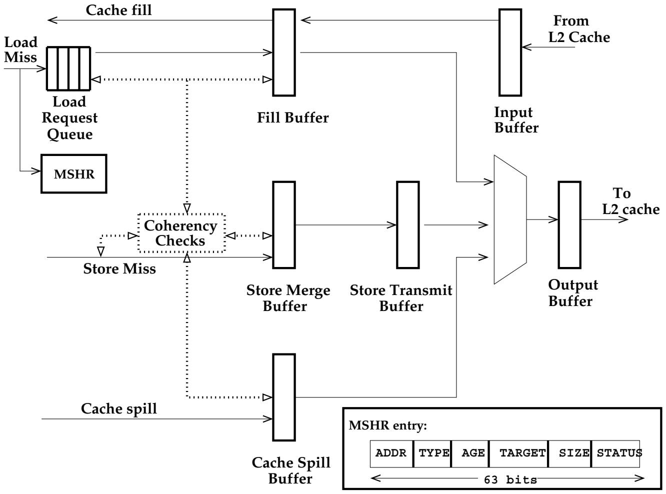


Fig. 6. Block diagram of the MHU. Inset shows the logical structure of the MSHRs

B. *MHU Operations*

The MHU is capable of filtering redundant read requests (for load misses) and coalescing smaller write updates (store misses) into larger chunks before sending them out into the OCN. Both of these optimizations are critical to improving the packet efficiency and utilization of the OCN.

Load Miss Processing: On a cache load miss, the MHU allocates a MSHR to hold the information pertaining to the load. If there are no pending requests to the same cache line the load is placed in the LRQ . To avoid deadlock conditions, the pipelines in the DT ensure that requests accepted in the DT can always be allocated in the MSHRs and the LRQ. When a free fill buffer entry becomes available, and the OCN port is available, a fill buffer is allocated for the transaction, the load is removed from the request queue and a read request is sent on the OCN.

When a read reply returns on the OCN, the reply data is placed in the fill buffer allocated for that transaction. As the data arrives from the OCN, the load(s) corresponding to the missed data are identified in the MSHR, packaged as if it were a load entering the DT for the first time and sent to access the caches and the LSQ. This approach of re-injecting the load as new loads significantly simplifies bypassing between stages and correctness reasoning in the MHU.

Store Coalescing: A OCN transaction requires one header flit and between one and four data flits depending on the size of datum. To minimize the overhead of header packets, the MHU attempts to create larger packets by coalescing multiple stores misses to same cache line. This functionality is provided by the merge and transmit buffers in the MHU. If the L1 store miss is to the same cache line as the cache line currently in the store merge buffer then the missed store updates the merge buffer. If the store miss is to a different cache line, then the line in the merge buffer is moved to the transmit buffer and the new store is allocated in the merge buffer. Once a line is moved into the transmit buffer, the MHU logic scans and packages the cache line into fewest possible flits.

MHU Coherence Policies: As the MHU handles both L1 load and store misses concurrently, it needs to ensure coherency between load and store misses to the same address. The load misses and store updates to the same cache line can arrive in three different orders at the MHU: 1) the store update arrives before the load request to the same address, 2) the store update arrives after the load request, or 3) both the load request and store update arrive at the same time.

In case 1, instead of forwarding to the load from the merge buffers, the merge buffers are flushed to the memory system. The load is then issued to the memory system as usual. This strategy takes avoids building complex forwarding logic between the merge buffer and the incoming load request for an uncommon execution scenario. In case 2, the strategy adopted for case 1 will not work because the load request may have already read the L2 caches and outside of the L1 none of the structures have store-to-load forwarding capabilities. In this case, the store updates the corresponding bytes of the matching fill buffer entry. When the load data arrives, it ensures that the store update bytes are not overwritten. In case 3, the store update is held back a cycle so that it becomes a write-after-read case as described in case 2.

C. Design Rationale

Why fill buffers? For deadlock-free operation, the MHU should not have any resource dependences on the OCN. Namely, the MHU can never refuse to accept incoming OCN replies while waiting for the OCN to accept new outgoing requests. Fill buffers guarantee deadlock-free operation by providing support for decoupling the OCN fills from rest of the MHU as they are always allocated prior to generation. Another alternative is to directly fill from the OCN into the caches. This strategy requires either a dedicated cache port for fills or mechanisms to premptively acquire cache ports. The former is undesirable because of area constraints and latter because of the high complexity.

Why Load Request Queues? LRQ's are used to improve the utilization of the fill buffers and to avoid conservatively stalling on mergeable misses. For deadlock-free operation, two slots have to be reserved in all MHU load handling structures, so that loads already in the pipeline can be slotted. Pre-reserving two slots in the fill buffers can be constraining because there are only four fill buffers. Providing more fill buffers is expensive in terms of area and restricted by timing constraints. To work To work around this, the fill buffer allocation is decoupled from the load execution by using the request queues and pre-reserving slots in the request queue. Pre-reserving slots is area-efficient and timing-efficient because LRQ is smaller than fill buffers (45 bits vs 512 bits).

Sizing of structures: The number of MHU entries was chosen so as to saturate the link between the DTs and the OCN. For uninterrupted traffic on the link, the MHU should have sufficient MSHRs to hold all incoming loads between the L1 miss detection and L1 reply for a load. Assuming one load is issued every cycle in this interval, and given that the average round trip for miss handling is 14 cycles, we would need fourteen MSHRs. An additional two MSHRs are required for deadlock-free operation, bringing the total number of MSHRs to sixteeen. If all these loads are to different cache lines, then a 16-entry fill buffer is required. However, to meet cycle times and constraints of our ASIC methodology we restricted the number of fill buffers to four. This is unlikely to be performance critical because load misses are commonly clustered and contiguous and hence it is uncommon to have 16 back-to-back loads to different cache lines to one DT partition.

VI. Physical Design

Each DT has an 8KB, 2-way set associative cache with 64 byte lines. The cache has 1 read and 1 write port. Each LSQ bank contains 256 entries and consists of CAM and RAM parts. The CAM is physically constructed out of eight 32-entry CAMs (one per block) and has a read, write and search port. The DTLB is a 16 entry, 48-bit wide CAM with two search ports (one load, one store), one read port and one write port. The predictor is built using a single ported 1024-bit array.

Area: Figure 7 shows the DT floorplan after synthesis, timing and layout optimizations. The design was implemented using IBMs 130nm ASIC process and the DT measures 3.37mm × 1.188mm. The CAM is entirely synthesized from flip-flops and occupies a large fraction of the DT area. A Custom CAM is likely to be smaller than the synthesized CAM, but our design methodology did not support integration of custom CAMs into the design flow.

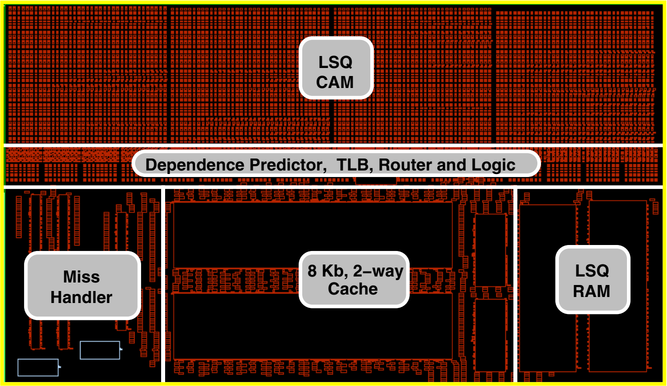


Fig. 7. Major structures in the DT floorplan

Timing and Critical Paths: The design synthesized to 3.2ns under worst case process parameters and operating conditions. The top critical path is the detection of store forwarding in the LSQ and the generation of signals for stalling the DT pipelines during store forwarding. At a high level, this process involves generating a eight-bit mask that encodes the blocks older than the load, ANDing with another eight-bit mask that encodes blocks with matching stores, and then performing a cumulative OR on the resulting 8-bit mask. This process takes up roughly 45% of the cycle time. Then, the forwarding signal must be distributed to rest of DT pipelines to stall conflicting operations. The high fanout on this signal contributes significantly (27% of cycle time) to the total delay.

The next most critical path is the logic for extracting and packaging store misses to the L2. There are two components to the delay, the first involves the actual extraction process and the second that is a stall signal that is asserted when a multi-cycle OCN transaction has to be generated. For higher frequency implementations, adding an extra stage to the pipeline can eliminate the critical path, without affecting performance because the store writes are unlikely to be critical operations.

The third most critical path involves checking for coherence in the MHU and performing coherence updates. The source of the problem is not the delay associated with the coherence checks, but the late availability of the load address that drives the coherence checks. Speculatively performing the coherence checks a cycle earlier can result in false positives which can complicate the design. Adding another stage (for the uncommon case of a coherence violation) in the pipeline will reduce performance by increasing the latency of missed loads.

VII. Concluding Remarks

The TRIPS microarchitecture includes a primary memory system that is *fully partitioned* and capable of supporting *high levels of memory level parallelism*. The memory system is made up of four Data Tiles(DT), partitioned by interleaving based based on addresses of the memory instructions. To support high levels of memory level parallelism, the DT utilizes memory side dependence predictors, deep LSQs, and an agressive miss handling unit capable of supporting up to 16 outstanding load misses per DT (64 per core). The design, implementation and verification required 21 person months and our design experience suggests that the design complexity of the partitioned memory system is comparable with the complexity of a centralized memory system.

A completely partitioned memory system like TRIPS provides a complexity-effective way of increasing the capacity and bandwidth of the memory system by simply increasing the number of partitions. For instance, eight loads/stores per cycle can be supported on TRIPS with eight DTs. However, for such partitioning to be beneficial and feasible (1) the memory instructions should be placed close to the cache banks to which their addresses map to, (2) the area overheads from replicated structures like LSQs should be minimized, and (3) the mechanisms used for communicating across the partitions should scale.

The TRIPS compiler team is currently investigating techniques for placing the memory instructions closer to DTs by array alignment analysis and sophisticated profile driven optimizations. In follow-up research, we have solved the area overhead problem, by utilizing the on-chip network for safely buffering overflows from the undersized LSQs [10]. Efficient store-tracking mechanism for sixteen or more partitions is a challenging problem and left for future work. Solutions to these problems are the last few remaining steps towards scalable and completely distributed memory systems.

VIII. Acknowledgments

We thank the anonymous reviewers for their suggestions that helped improve the quality of this paper. This research is supported by the Defense Advanced Research Projects Agency under contracts F33615-01-C-4106 and NBCH30390004 and an NSF instrumentation grant EIA-0303609.

References

[1] Elliot Waingold et al., “Baring it all to software: RAW machines,” *IEEE Computer*, vol. 30, no. 9, pp. 86–93, September 1997.

[2] Doug Burger et al., “Scaling to the End of Silicon with EDGE architectures,” *IEEE Computer*, vol. 37, no. 7, pp. 44–55, July 2004.

[3] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, “WaveScalar,” in *Proceedings of the 36th Micro*, December 2003, pp. 291–302.

[4] G. S. Sohi and M. Franklin, “High-bandwidth data memory systems for superscalar processors,” in *Proceedings of the 4th ASPLOS*, Apr. 1991, pp. 53–62.

[5] L. Ceze, J. Tuck, and J. Torrellas, “Are we ready for high memory-level parallelism?” in *Proceedings of the 4th Workshop on Memory Performance Issues*, February 2006.

[6] S. Vetter, S. Behling, P. Farrell, H. Holthoff, F. O’Connell, and W. Weir, *The POWER4 Processor Introduction and Tuning Guide*. IBM, 2001.

[7] D. Kroft, “Lockup-free instruction fetch/prefetch cache organization,” in *Proc. of the 8th annual symposium on Computer Architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, pp. 81–87.

[8] G. Z. Chrysos and J. S. Emer, “Memory dependence prediction using store sets,” in *Proc. of the 25th annual symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 142–153.

[9] *Intel Architecture 32 Family Developer’s Manual, Volume 3, Appendix A.2*. Intel, 2001.

[10] S. Sethumadhavan, D. Burger, and S. W. Keckler, “Partition the Banks, not the functionality, of Large-Window Load-Store Queues,” Department of Computer Sciences, The University of Texas at Austin, Tech. Rep. TR-06-39, 2006.