Presented at the 12th Symposium on Computer Architecture and High-Performance Computing, October 2000.

Efficient ECC-Based Directory Implementations for Scalable Multiprocessors

Kourosh Gharachorloo, Luiz André Barroso, and Andreas Nowatzyk

Western Research Laboratory
Compaq Computer Corporation
250 University Ave.
Palo Alto, CA 94301
{kourosh,barroso,agn}@pa.dec.com

Abstract

With increasing chip densities, next-generation microprocessor designs have the opportunity to integrate many of the traditional system-level modules onto the same chip as the processor. This integration changes some of the design trade-offs for how and where to store directory information. One extremely attractive option is to support directory data with virtually no memory space overhead by computing memory ECC at a coarser granularity and utilizing the unused bits for storing the directory information. Compared to providing a dedicated memory and datapath for directory storage, this approach leads to lower cost and a simpler design by requiring fewer components and pins. Furthermore, this approach leverages the low latency, high bandwidth path to memory provided by the integration of memory controllers onto the processor chip. However, without careful design, maintaining data and directory bits together can lead to potential inefficiencies in the form of extra memory bandwidth usage and memory controller occupancy, and extra memory latency.

This paper describes the techniques used in the context of the Piranha design [3] to provide an efficient ECC-based directory implementation which addresses the occupancy/bandwidth and latency issues. Our approach for dealing with the occupancy/bandwidth issues involves either eliminating the extra read and write operations or performing partial memory accesses (instead of accessing the whole block). This is achieved by a combination of techniques which include (i) augmenting the L2 caching state to keep track of some critical directory state, (ii) making up dummy data for protocol transactions with a stale memory copy, and (iii) maintaining a partial ECC that is used to compute the combined ECC of the data and the modified directory bits without needing the actual data bits. To address the latency issues, we replicate critical directory state in different segments of the memory line which allows us to efficiently support the critical-word-first optimization by pipelining data from memory to the requester before all the data is read from memory. The combination of the above techniques also eliminates all the inefficiencies that arise due to maintaining a combined ECC for directory and data bits. Therefore, we benefit from the more efficient use of bits provided by the combined ECC with virtually no performance penalty compared to maintaining separate ECC bits for data and directory. Finally, the optimizations used in Piranha are general and applicable to other designs that use ECC-based directories.

I. INTRODUCTION

Advances in semiconductor technology enable next-generation microprocessor designs with well over a hundred million transistors on a single die. At the same time, the increasing density and speed of transistors make off-chip communication relatively more expensive. These technology trends provide an incentive to integrate functional modules that traditionally appear at the system-level onto the same die as the processor [4]. For example, the next-generation Alpha 21364 plans to aggressively exploit such integration by including a 1GHz 21264 core, two levels of caches, memory controllers, coherence hardware, and network router all on a single die [2]. The tight coupling of these modules enables a more efficient and lower latency memory hierarchy which can substantially improve application performance [4]. Furthermore, integrated designs such as the Alpha 21364 provide glueless scalable multiprocessing, whereby a large server can be built in a modular fashion using only processor and memory chips.

The integration of the coherence hardware and memory controllers on the same chip leads to interesting design choices for how and where to store directory information. One extremely attractive option is to support directory data with virtually no memory space overhead by computing memory ECC at a coarser granularity and utilizing the unused bits for storing the directory information [8][9]. Given the trend towards larger main memories, dedicated directory storage can become a significant cost factor. Similarly, providing a dedicated datapath for a separate external directory memory reduces the number of pins available for supporting memory and interconnect bandwidth. Therefore, compared to having a dedicated external storage and datapath for directories, the ECC-based directory approach leads to lower cost by requiring fewer components and pins, and provides simpler system scaling since directory entries naturally scale with memory size. Furthermore, this approach leverages the low latency, high bandwidth path provided by the integration of memory controllers on the chip for both memory and directory access.

Figure 1 shows various granularities at which ECC can be computed for a 64-byte line, and the number of unused bits that can be used for storing directory information in each case. The figure assumes a memory width of 144 bits, with 128 bits for data and 16 bits available for ECC. Computing ECC across 64-bit segments (as is typically done) uses up all 16 bits. However, computing ECC across

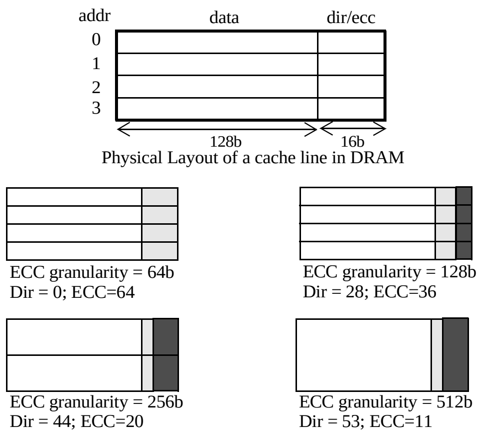

Figure 1. Data, directory, and ECC layout in standard 144-bit wide SDRAMs (64B block).

256-bit segments renders 44 extra bits for encoding directory information for a 64-byte line. The ECC is encoded to cover both the data and directory bits in a combined fashion. Maintaining separate ECC for data and directory is less efficient and leads to fewer unused bits (32 bits vs. 44 bits in the example above). Finally, as shown in the figure, the ECC and directory bits are spread across various segments of the line; this allows for standard width memory modules (e.g, 72 or 144-bit) while maintaining regular alignment for the data bits.

Maintaining data and directory bits together works well for many coherence transactions where both the data and directory need to be read or written. In such cases, the read or write to the directory effectively comes for free as part of the data operation. However, there are some important coherence transactions where only the directory needs to be accessed (e.g., when data is held exclusive at a remote node). These latter cases can lead to extra occupancy and bandwidth usage at the memory controllers compared to a system with separate data and directory storage. Furthermore, even though on-chip memory controllers provide fast access to memory (e.g., 40-60ns), the fact that the whole memory line must be read to construct the complete directory state leads to potential latency concerns, especially when a read of the data bits is not necessary for servicing a transaction.

This paper describes the techniques used in the context of the Piranha design to provide an efficient ECC-based directory implementation which addresses the occupancy/bandwidth and latency issues described above. Piranha [3] is a research prototype being developed at Compaq that aggressively exploits chip multiprocessing by integrating eight simple Alpha processor cores, separate instruction and data caches for each core, a shared second-level cache, eight memory controllers, two coherence protocol engines, and a network router all on a single die. With respect to memory controller *occupancy and bandwidth issues*, our goal is to either (i) eliminate extra read and write operations, or (ii) do partial instead of full read or write operations. With respect to the *latency issues*, our goal is to allow critical directory state (e.g., whether a line is clean at memory or exclusive at a remote node) to be constructed without requiring a full read.

Our approach for eliminating some of the extra read and write memory operations involves *augmenting the L2 caching state* to keep track of whether data is solely cached at the home node or is also being cached by remote nodes. For local transactions at the home node, the fact that data is solely cached at the home node can be used to eliminate reading, and sometimes writing, the directory information in memory. Similarly, writes to the directory are eliminated given the directory only maintains caching information for remote nodes. Furthermore, remote transactions that arrive at the home node first check the L2 before accessing memory. In cases where the L2 at the home node has the sole copy, we eliminate the memory read since the L2 can provide the data and the directory can be inferred to be null. Finally, the fact that the L2 at the home node is caching a given data (whether a sole copy or not) can also be used to eliminate memory reads in a few protocol race cases.

For transactions where the L2 cannot eliminate extra memory accesses, we attempt to use partial read accesses to obtain sufficient directory information and to do partial modifications to directory whenever possible. The cases where partial accesses suffice depend on both protocol characteristics and the directory representation. However, there are a couple of issues that complicate using partial accesses. The *first issue* arises from the fact that the ECC bits cover both data and directory information. Therefore, transactions that do full modifications of directory potentially require a full data read for computing the combined ECC even though a partial read of the line would have provided sufficient directory state for servicing the transaction. The *second issue* arises because the read of the data and/or directory at the memory controller is separated in time from the write to the directory. This fact, along with the need to compute a combined ECC for the directory write, leads to the potential need for either (i) reading the data again at the time of the directory write, or (ii) buffering the data that is read at the memory controller until the directory write arrives.

For many of the transactions that are affected by the above two issues, it turns out that the data in memory is rendered stale as part of servicing the transaction (e.g., an exclusive copy is given to a remote node). In such cases, we can avoid the inefficiencies by simply making up dummy data (e.g., all zeros) to compute the combined ECC to be used with the directory write (*DDW*, for *dummy data write*). However, the DDW optimization cannot be used when the data in memory is not stale. For such transactions, we eliminate the inefficiencies arising from the second issue above (separation in time of read and write to directory) by maintaining a *partial ECC* (*PECC*) for the data bits when they are initially read. At the directory write time, this partial ECC is used to compute the combined ECC of the data and the modified directory bits without needing the actual data bits. The DDW and PECC optimizations together eliminate all the inefficiencies that arise due to maintaining a combined ECC for directory and data bits. Therefore, we benefit from the more efficient use of bits provided by the combined ECC with virtually no performance penalty compared to maintaining separate ECC bits for data and directory.

Finally, our approach for latency issues is to allow critical directory state to be constructed without requiring a full read of the memory line. To achieve this, we *replicate direc-*

tory state bits in each segment of the memory line (e.g., the Piranha design has two 256-bit segment, with 2 bits of directory state per segment). This technique is especially important in light of the critical-word-first optimization which leads to a different starting segment for memory read operations depending on the address specified by the processor request. The replication of directory state allows a partial read to any segment to provide sufficient information for quickly determining whether the data in memory is up-to-date, allowing us to pipeline the data transfer to the requester before the whole line is read from memory. A potential shortcoming of directory state replication is that all replicated state bits must be modified when the directory state changes, possibly incurring more read or write bandwidth. As it turns out, our protocol incurs virtually no extra read bandwidth due to this replication. And even for designs that may incur some extra bandwidth, the benefits of latency reduction from this technique can easily outweigh the increased memory controller occupancy.

The optimizations used in Piranha are general and applicable to other designs that use ECC-based directories (with a few of the optimizations applicable to any directory-based scheme). The next section provides an overview of the Piranha architecture and the basic issues relevant to ECC-based directories. Section 3 describes the techniques we use to make ECC-based directories more efficient.

II. Piranha System Overview

Piranha [3] is a research prototype being built at Compaq (a collaboration between Corporate Research and the NonStop Hardware Development group) to explore the design space of scalable systems based on chip-multiprocessors (or CMPs). This section presents a brief overview of Piranha. For a more detailed description of the Piranha architecture, please refer to Barroso et al. [3].

Similar to the next-generation Alpha 21364 [2], Piranha integrates on-chip functionality to allow for scalable shared-memory multiprocessor configurations to be built in a glueless and modular fashion. The centerpiece of the Piranha architecture is a highly-integrated *processing node* (referred to as CMP chip in Figure 2) with eight simple Alpha processor cores, separate instruction and data caches for each core, a shared second-level cache, eight memory controllers, two coherence protocol engines, and a network router all on a single die. The Piranha design also includes an *I/O node* which reuses all the modules from the processing node, but has only one Alpha CPU and one-eight of the L2 and memory capacity. The I/O node also includes a PCI interface for connecting to I/O devices.

Figure 2 shows the block diagram of a Piranha CMP chip. Each *Alpha CPU core (CPU)* is directly connected to dedicated instruction (*iL1*) and data cache (*dL1*) modules. These first-level caches interface to other modules through the *Intra-Chip Switch (ICS)*. On the other side of the ICS is a logically shared *second level cache (L2)* that is interleaved into eight separate modules, each with its own controller and on-chip tag and data storage. Attached to each L2 module is a *memory controller (MC)* that directly interfaces to one bank of DRAM chips. Each memory bank provides a bandwidth of 1.6GB/sec, leading to an aggregate bandwidth of 12.8 GB/sec. Also connected to the ICS are two protocol engines, the *Home Engine (HE)* and the *Remote Engine (RE)*, which support shared memory across multiple Piranha

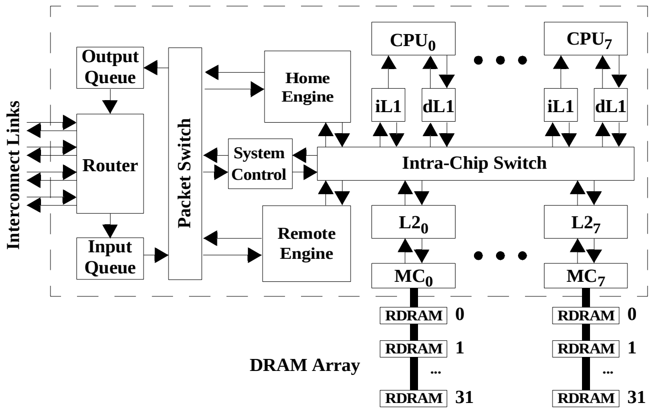

Figure 2. Block diagram of Piranha's CMP chip

chips. The interconnect subsystem that links multiple Piranha chips consists of a *Router (RT)*, an *Input Queue (IQ)*, an *Output Queue (OQ)* and a *Packet Switch (PS)*. The total interconnect bandwidth (in/out) for each Piranha processing chip is 32 GB/sec.

A Piranha system consists of a mix of CMP, I/O, and DRAM chips, connected in any arbitrary topology (Figure 3). All the memory in the system is logically shared (despite being physically distributed), and all caches are kept coherent by hardware. In the following sections we briefly overview Piranha's coherence scheme and DRAM organization in order to provide the necessary background for the techniques proposed in this paper.

A. Coherence Protocol

Piranha enforces coherence both among the many L1 caches within a CMP chip as well as across chips. Responsibility for enforcing coherence is split between the L2 cache controller and the protocol engines (HE and RE), with the L2 cache maintaining intra-chip coherence and interfacing with the memory controller, and the protocol engines being responsible for inter-chip coherence and interfacing with the interconnect subsystem. The *home engine (HE)* is responsible for exporting memory whose home is at the local node, while the *remote engine (RE)* imports memory whose home is remote.

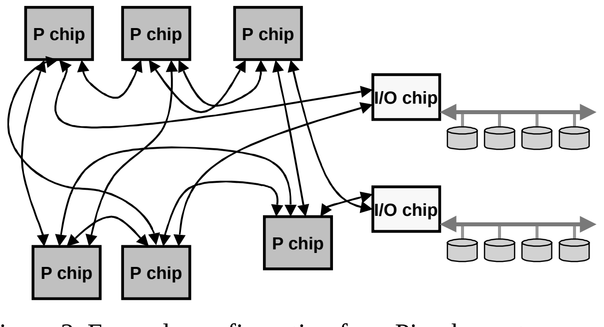

Figure 3. Example configuration for a Piranha system with six CMP (8 CPUs each) and two I/O chips.

The L2 keeps state and tags for all blocks that are currently cached locally, providing complete and exact information about sharing within one chip. All L1 accesses and accesses from remote nodes are first sent to the L2, which may respond directly, access the local memory, or forward the request to another L1 or to one of the protocol engines.

The inter-node coherence protocol, implemented by the HE and RE, uses an invalidation-based directory protocol with support for four request types: *read* (*RD*), *read-exclusive* (*RDEX*; i.e., a write miss), *exclusive* (*EX*; requesting processor already has a shared copy), and *exclusive-without-data*[1] (*ITOD*). We use two different directory representations depending on the number of sharers: limited pointer (LP) [1] and coarse vector (CV) [5]. Two bits of the directory are used to encode four possible states: *uncached* (*UNC*; no remote node has a copy), *shared-LP* and *shared-CV* (*SHR*), and *exclusive* (*EXCL*). The interpretation of the remaining directory bits (the *sharer bits*) is based upon the directory state. The directory is not used to maintain information about sharers at the home node. Furthermore, directory information is maintained at the granularity of a node (not individual processors). Given a 1K node system, we switch to coarse vector representation past 4 remote sharing nodes.

B. DRAM and Directory Organization

The implementation of the memory directory is a critical design decision in NUMA systems. The SGI Origin2000 [7] uses a separate off-chip SRAM bank for directory storage. This solution is undesirable for mainly two reasons: it increases system cost and parts count, and it makes upgrading complex since directory SRAM chips have to be added when more DRAM or processors are added to the system. In addition, in a *system-on-a-chip* design, such as Piranha and the Alpha 21364, a separate SRAM channel would consume extra pins (which are at a premium in such systems) and chip area for the SRAM controller logic.

An interesting alternative, initially proposed for the S3.mp prototype [8] and also used by the Sun UltraSPARC-III [6], is to store the memory directory in the same DRAM bank in which the data is kept by changing the granularity in which ECC is calculated, as previously shown in Figure 1. Typical industry standard DRAM banks include 8 bits of ECC for each 64 bits of data. If we calculate ECC over 128 bits, the ECC requires only 9 bits, leaving 7 bits free for other uses. With a 64B block, this yields a total of 28 extra bits that can be used to store the directory (this is the scheme used by UltraSPARC-III). In Piranha, we calculate ECC over 256 bits, freeing up 44 bits for the memory directory.

Note that in both UltraSPARC-III and Piranha the same ECC is covering both the data and the memory directory bits. Using separate ECC coverage for directory and data may simplify directory manipulation, but dramatically reduces the number of bits available for use as directory and can impact the scalability of the system. For example, computing separate ECC for the directory and data with a data ECC granularity of 128b leads to only 21 bits for the directory (7 bits are used for the directory ECC), compared to 28 bits in the combined ECC case. Table I depicts the maximum number of data bits covered by a Single-Error-Correction, Double-Error-Detection (SECDED) code as the number of SECDED bits increases.

TABLE I
MAXIMUM NUMBER OF BITS COVERED BY AN SECDED CODE

No. of SECDED bits	5	6	7	8	9	10	11
Max no. of data bits covered	11	26	57	120	247	502	1013

Combining data and directory in the DRAM banks through coarser granularity ECC effectively results in zero storage overhead for memory directory, and also avoids the need for a separate set of pins and controller logic to access the memory directory. The main drawback is reduced transient bit-error coverage, which is offset by the decreasing soft-error rates in newer DRAM chips and by using simple techniques such as periodic DRAM scrubbing[2].

III. Techniques for Efficient Directory Access

Despite the many advantages of using a combined ECC encoding for data and directory, a straightforward implementation of this approach can lead to higher latencies and inefficient use of the DRAM channel. In this section we describe the inefficiencies of the straightforward implementation, and present a set of techniques that either partially or fully eliminates them.

Maintaining data and directory bits together (with a combined ECC) works well for many coherence transactions where both the data and directory need to be read or written. In such cases, the read or write to the directory effectively comes for free as part of the data operation. However, there are some important coherence transactions where only the directory needs to be accessed (e.g., when data is held exclusive at a remote node). These latter cases can lead to extra occupancy and bandwidth usage at the memory controllers compared to a system with separate data and directory storage. Furthermore, even though on-chip memory controllers provide fast access to memory (e.g., 40-60ns), the fact that the whole memory line must be read to construct the complete directory state leads to potential latency concerns.

We provide three types of techniques to deal with the above inefficiencies. First, as described in Section 3A, we eliminate some extra memory operations by augmenting the L2 cache state to keep track of whether data is exclusively cached at the home node. Second, we attempt to reduce memory bandwidth usage by issuing partial read and write accesses whenever possible. However, there are a number of issues that complicate the use of partial accesses. Section 3B describes the techniques to allow the use of partial accesses despite these complications. Finally, Section 3C describes our solution to the latency issues which allows critical directory state to be constructed without requiring a full read of the memory block.

For simplicity, we assume the Piranha memory system parameters for the discussion in this section. As mentioned before, Piranha uses 64B cache lines (or blocks), a 144b-wide DRAM channel, and an ECC granularity of half a cache line (256b data + 22b directory). Furthermore, as in

[1] This corresponds to the Alpha write-hint instruction (wh64) which indicates that the processor will write the entire cache line, thus avoiding a fetch of the line's current contents (e.g., useful in copy routines).

[2] Scrubbing is a common feature in modern DRAM controllers and consists of issuing periodic memory reads to check the ECC and correct single-bit errors in order to decrease the likelihood of double-bit errors (which are uncorrectable with the above mentioned SECDED ECC codes).

Piranha, we assume that accesses from remote nodes first check the L2 at the home node before checking the directory.

A. *Eliminating Memory Accesses by Augmenting L2 Cache States*

A simple optimization that saves reads and writes to the directory is to augment the state in the L2 to keep track of cases where the home node is the only one currently caching a block. This is accomplished by maintaining a notion of "node exclusiveness" as part of the L2 state[3], and by not using the directory to maintain information about sharers at the home node. The fact that the directory does not keep track of caching at home does not affect correct operation since accesses from remote nodes always have to check the L2 state. Furthermore, data that is only accessed at home does not incur the cost of writing the directory.

The basic idea is to mark the state in the L2 as node exclusive every time a read is issued by a home L1 and the directory state is UNC, or whenever a home L1 issues a write. Table II lists all the cases in which just by checking the L2 state, directory accesses are avoided for L1 requests at the home node. The first column of the table lists the protocol request types described in Section 2A, in addition to the *write-back* (*WB*) and *replacement* (*REPL*) requests. A WB is issued when a dirty exclusive copy is replaced (and therefore needs to write the block to memory). A REPL is issued when a clean exclusive copy is replaced. The REPL transaction does not carry a copy of the block since it only needs to update the memory directory.

TABLE II
DIRECTORY ACCESSES AVOIDED FOR LOCAL HOME REQUESTS WHEN THE L2 AT HOME HAS A COPY OF THE BLOCK

Local request	Home L2 state	Dir Read Avoided?	Dir Write Avoided?
RD	valid	yes	yes
	node excl.	yes	yes
RDEX	valid	no	yes[1]
	node excl.	yes	yes
EX	valid	no	yes[1]
	node excl.	yes	yes
ITOD	valid	no	yes[1]
	node excl.	yes	yes
WB	valid	--	--
	node excl.	yes	no[2]
REPL	valid	--	--
	node excl.	yes	yes

1. A dir. write is not needed since the next L2 state will be node exclusive. The dir. write is avoided even if the L2 state was invalid.
2. Dir. write is needed, but it incurs no overhead since the full data has to be written as well.

As the table shows, directory reads and writes can be avoided in most cases. The exceptions are the directory reads for RDEX, EX, and ITOD requests when the node does not have an exclusive copy, since in these cases it may be necessary to invalidate remote sharers. The directory write for a WB when the L2 is node-exclusive is not avoided by this optimization, but it incurs no overhead since a full data block write is also needed.

[3] From this point on we refer to L2 state as the combination of state information that is kept in the Piranha L2 controller, which includes state/tags for blocks that reside in the L2 itself as well as duplicate state/tag information for all the L1 caches in the chip.

TABLE III
DIRECTORY ACCESSES AVOIDED FOR REMOTE REQUESTS WHEN THE L2 AT HOME HAS A COPY OF THE BLOCK.

Remote request	Home L2 state	Dir Read Avoided?	Dir Write Avoided?
RD	valid	no	no
	excl.	yes	no
RDEX	valid	no	no
	excl.	yes	no
EX	valid	no	no
	excl.	yes	--
ITOD	valid	no	no
	excl.	yes	no
WB	valid	yes	--
	excl.	yes	--
REPL	valid	yes	---
	excl.	yes	---

The node-exclusivity optimization is only beneficial when the L2 is caching the data due to some local processor having accessed it. However, once the L2 caches this data, accesses from remote nodes can also benefit from this technique. Essentially, whenever the home L2 is node-exclusive, all directory reads are avoided since the L2 can infer that the directory indicates no remote sharers and an UNC state. Table III lists all cases in which a directory access (read or write) is avoided for remote requests by using L2 cached information. Essentially a directory read is avoided when the home L2 has a node-exclusive copy. However, directory writes are always needed. Note that in all cases in which a directory read is avoided both DRAM occupancy and request latency are reduced, since both the data and the (inferred) directory can be supplied by the L2 without having to wait for a DRAM read cycle.

This scheme can be seen as a restricted form of directory caching [5]. Adding an actual directory cache could further improve efficiency of coherence maintenance at the cost of additional hardware and complexity

B. *Addressing Memory Occupancy Penalties*

To better understand the DRAM channel occupancy penalties, consider a straightforward implementation in which the following actions take place for every protocol request:

1. Memory directory is read from DRAM and shipped by the L2 to the Home Engine (HE).
2. HE sends appropriate messages, waits for possible acknowledgments, and updates the directory.
3. HE ships the updated version of the directory to the L2 which issues a directory write operation to the DRAM controller.

In the sequence above, a full data block read is performed for all coherence transactions, even in cases where data is not needed, or in which only parts of the directory would have been relevant to the particular transaction. We refer to this initial full block read as *memRD1*. In step 3, a full data block write is performed (labeled *memWR*) even though in the vast majority of the cases only the directory needs to be updated. More seriously, if in step 3 only the directory is being updated, the data value is necessary to calculate the correct ECC bits. In order to perform the write, either (i) the data has to be read again from memory, transforming this write into a read-modify-write operation, or (ii) the data block must be buffered at the L2 controller between step 1 and step 3. The first option is prohibitively wasteful of DRAM channel bandwidth, given the cost associated with the extra read (labeled *memRD2*). The second option is also undesirable given that we want to support a large number of outstanding transactions, which would consume area-expensive block-sized buffers for the duration of all such transactions (this duration can be large if network replies are needed before the directory can be updated).

In the remainder of this section we present a set of techniques that completely eliminate memRD2, and either eliminate or reduce the size of memRD1 and memWR for many common protocol operations. The resulting system has comparable directory manipulation overheads to one that uses separate ECC encoding for data and directory, while allowing much better scalability due to the larger number of bits available for directory information when we use a combined ECC encoding.

B.1 Exploiting Partial Directory Reads/Writes

One way to save DRAM channel occupancy is to avoid reading or writing the full memory block whenever possible. The minimum amount of data that can be read or written is an ECC segment, which in our case is half of the 64B line (given that we compute ECC on 32B segments).

We identify the cases where partial reads and writes are possible in Section 3D. However, there are complications that limit the feasibility of partial accesses. The *first issue* arises from the fact that the ECC bits cover both data and directory information. Therefore, transactions that do full modifications of the directory potentially require a full data read for computing the combined ECC, even though a partial read of the line would have provided sufficient directory state for servicing the transaction. The *second issue* arises because the read of the data and/or directory at the memory controller is separated in time from the directory update (i.e., the time between steps 1 and 3, mentioned in the previous section). In what follows, we describe the two techniques used in Piranha that allow the use of partial accesses despite the above complications.

B.2 Dummy Data Writes (DDW)

We observe that in many cases in which a memRD2 read would be required to update the directory, the copy of the data in memory is stale. This arises whenever the incoming request is an EX, RDEX, or ITOD, since such accesses indicate to the home that the block will be modified by the remote node. In these cases, the memRD2 can be avoided by “making up” dummy data and combining it with the new directory to generate a valid ECC (referred to as *dummy data writes*, or *DDW*).

This technique can eliminate memRD2 reads in all cases except when the current value of the data in memory is not stale. These remaining cases consist of RD requests when the directory state is UNC or SHR, and the REPL request since it changes the directory state to UNC from EXCL but (unlike WB) does not carry data with it.

The DDW technique also enables a partial memRD1 for transactions that would need a full memRD1 otherwise because of the need to update the whole directory.[4]

B.3 Caching of Partial ECC (PECC)

To deal with the cases where DDW cannot be used, our design uses a special ECC code in which a *partial ECC* (*PECC*) computed on the data only can be used along with the directory to generate the combined ECC (there are many SECDED ECC codes that meet this requirement). By doing this, we can avoid the memRD2 read by computing the PECC on the data at the time of memRD1 (step 1 in Section 3B) and caching it with the pending transaction buffer in the L2 that is keeping track of the outstanding request. Caching the PECC is much less costly than buffering the entire data since the PECC is fairly small (20 bits in Piranha). At the time of the directory update, a masked write operation is performed in which only the directory and ECC bits are stored.

The PECC technique eliminates the remaining memRD2 cases not handled by DDW. As we will see later, there are some protocol transactions where either DDW and PECC can be used to avoid memRD2.

C. Addressing Memory Latency Penalties

Using a combined ECC encoding for directory and data can hurt latency in several cases. One issue is that a full ECC segment (half a block in our case) must be read, and its ECC must be checked for correctness, before any part of that segment can be reliably used. Because of this, transferring data from memory to a requester incurs at least the store-and-forward delay of buffering one ECC segment. However, such buffering may already be required to address a bandwidth mismatch between the on-chip interconnect and the DRAM channel (this is the case in the Piranha design). Therefore, this store-and-forward delay is not solely caused by the granularity of the ECC encoding.

A more serious latency problem results from the need to implement critical-word-first ordering (CWF) of miss replies. CWF ordering is used in most high-performance CPUs, and consists of presenting the miss reply in an order such that the missed word is delivered first, allowing the CPU to resume execution earlier, and therefore reducing the exposed miss latency. However, the L2 controller needs to inspect the (2-bit) directory state before it can initiate a response. For instance, if the directory state indicates EXCL, the data in memory is stale and therefore should not be returned to the requester. If the CWF ordering of a particular request happens to first fetch the half-block in which the directory state resides, there is no further latency penalty. Unfortunately, given that CWF ordering is random, one can expect that 50% of the time the CWF half block will not be the one that contains the directory state bits. In this case, a full block store-and-forward penalty (as high as 30ns in high-end DRAM systems) will be incurred.

[4] The DDW optimization can also be applied to directory caching schemes. For schemes that actually write back the directory information on a replacement, DDW can be used to replace a read-modify-write with just a write operation in some cases.

TABLE IV

CASES IN WHICH JOINT ECC ENCODING OF DIRECTORY AND DATA INCUR EXTRA DRAM BANDWIDTH WITH RESPECT TO THE IDEAL CASE FOR REMOTE REQUESTS.

Remote Request	Directory state	memRD1 overhead	memRD1 opt.	memWR overhead	memWR overhead (state replication)	memRD2 opt.
RD	UNC	0		1/2	1	PECC
	SHR (sharers fit in 1/2 dir)	0		1/2	1/2[1]	PECC
	SHR (otherwise)	0		1	1	PECC
	EXCL (critical word in 1st sub-block)	1/2		0	0	
	EXCL (critical word in 2nd sub-block)	1		0	0	
RDEX	UNC	0		1/2	1	PECC/DDW
	SHR (sharers fit in 1/2 dir)	0		1/2	1	PECC/DDW
	SHR (otherwise)	0		1	1	PECC/DDW
	EXCL (critical word in 1st sub-block)	1/2		1/2	1/2	PECC/DDW
	EXCL (critical word in 2nd sub-block)	1		1/2	1/2	PECC/DDW
EX	UNC	1/2		0	0	
	SHR (sharers fit in 1/2 dir)	1/2	DDW	1/2	1	DDW
	SHR (otherwise)	1		1	1	PECC/DDW
	EXCL	1/2		0	0	
ITOD	UNC	1/2	DDW	1/2	1	DDW
	SHR (sharers fit in 1/2 dir)	1/2	DDW	1/2	1	DDW
	SHR (otherwise)	1		1	1	PECC/DDW
	EXCL	1/2		1/2	1/2	PECC/DDW
WB	UNC	1/2		0	0	
	SHR	1/2		0	0	
	EXCL	1/2		0	0	
REPL	UNC	1/2		0	0	
	SHR	1/2		0	0	
	EXCL	1/2[2]		1/2	1	PECC

1. The first RD request that changes the directory mode from limited pointers to coarse vectors requires a full block write with state replication since the directory state actually changes from SHR-LP to SHR-CV.
2. With state replication, a full block read is needed because the directory write cannot use DDW for the second half of the block.

We address the above issue by guaranteeing that the 2-bit directory state will be available early, regardless of CWF ordering. To achieve this we *replicate the directory state* bits in all ECC segments (e.g., Piranha replicates the 2-bit state in the two ECC segments).

The state replication optimization improves miss latencies in many cases, but it reduces the number of cases where a partial memWR would be sufficient for modifying the directory. This is because both copies of the state bits need to be updated for any directory state changes, leading to a full memWR. Nonetheless, the benefits of the lower miss latency made possible by state replication can easily outweigh any negative effects of the increased memWR bandwidth.

D. Putting It All Together

Table IV enumerates all the cases in which our optimizations eliminate all or part of memRD1 or memWR operations. memRD2 accesses are not shown in the table since the combination of DDW and PECC eliminate them completely; the right most column of the table identifies which optimization(s) enable the elimination of memRD2.

The table enumerates all possible remote requests in the coherence protocol, and all possible directory states. In the memRD1 and memWR overhead columns, we indicate extra DRAM occupancy caused by accessing the directory relative to a design that uses a dedicated directory storage. The memWR case is split into two columns in order to identify the cases where the state replication optimization can cause extra memWR overhead. These overheads are measured as the fraction of a full block access that is required for the ECC-based scheme. Since we are using an ECC granularity of half a block, the smallest fraction of a block that can be accessed is one half. A zero (0) indicates no penalty for directory access (read or write), either because our optimizations make it unnecessary or because a data access is also needed (in which case the directory comes for free). A one-half (1/2) in the table indicates cases where half a block access (read or write) is needed in order to maintain the directory while the data access is not necessary. A one (1) in the table indicates that a full block access (read or write) is necessary in order to maintain the directory while the data itself is not needed. For each row, the table also indicates which of the DDW and PECC optimizations are enabling the partial block accesses in the case of memRD1, and eliminating the block read in the case of memRD2.

As shown in the table, many cases incur no overhead. RD and RDEX requests in which the directory state is UNC or SHR are such that the data is always required, therefore the directory read is free. Similarly, there is no memWR overhead for RD and WB requests that find the directory state EXCL[5] since in these cases a data writeback is also required. Finally memWR accesses are avoided in a variety of protocol race conditions in which the incoming request, given the current directory state, is considered stale and is consequently dropped. For example, an EX request that finds the memory state EXCL is dropped by the home since it can be concluded that another node has "raced" with this request and has already obtained exclusive ownership.

Overall the table shows that in the majority of the cases, our techniques can reduce the memRD1 overhead to only a half bock access. Similar reductions would also apply to the memWR overhead if not for the decision to replicate the directory state bits in both halves of the block. Despite its extra memWR overhead, state replication is still desirable since it often reduces miss latencies. Furthermore, the Piranha system is designed to have spare DRAM bandwidth to accommodate such overheads.

It is also interesting to note that the combination of our optimizations eliminate virtually all potential performance penalties compared to maintaining separate ECC bits for data and directory (primarily by eliminating the need for memRD2). Furthermore, the combined ECC approach leads to a larger number of bits available for directory storage.

IV. Concluding Remarks

The idea of computing memory ECC at a coarser granularity and utilizing the unused bits for storing the directory information is a compelling one since it enables scalable shared-memory multiprocessors with (i) virtually no memory space overhead for directory storage and (ii) natural scaling of directory entries with memory size. This approach becomes more compelling in next-generation designs whereby virtually all system-level modules are integrated onto a single chip. Such chip-level integration makes it even less attractive to provide a dedicated storage and datapath (and pins) for directory memory, especially in light of the fact that an ECC-based design can exploit the existing low latency, high bandwidth path to memory that is enabled by the integration. Nevertheless, achieving an efficient ECC-based directory design is non-trivial and requires a number of optimizations.

The Piranha design incorporates a number of novel optimizations to provide an efficient ECC-based directory design. These optimizations alleviate many of the issues related to extra memory bandwidth usage and occupancy and extra latency that can potentially accompany ECC-based designs. Furthermore, the combination of these optimizations allows us to maintain a combined ECC for directory and data bits (leading to a more efficient use of bits) while eliminating virtually all the potential performance penalties compared to maintaining separate ECC bits for data and directory. The techniques we use are applicable to any ECC-based directory design, with some techniques being applicable to any directory design. We hope the framework that is presented in this paper, along with the various optimizations described, leads to further research into ECC-based directory designs and the evaluation of various design trade-offs.

[5] For the WB case, not only the does directory state have to be EXCL, but the current owner node must be the issuer of the WB request. Otherwise this is a race condition, and a directory (or data) write is not performed.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Coherence. In *ACM Annual International Symposium on Computer Architecture*, pages 280-289, May 1988.

[2] P. Bannon. Alpha 21364: A Scalable Single-chip SMP. Presented at the *Microprocessor Forum '98 (http://www.digital.com/alphaoem/microprocessorforum.htm)*, October 1998.

[3] L. A. Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A Scalable Architecture based on Single-Chip Multiprocessing. In proceedings of the *25th Annual International Symposium on Computer Architecture, pages 282-293*, Vancouver, Canada, June 2000.

[4] L. A. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of Chip-Level Integration on Performance of OLTP Workloads. In *6th International Symposium on High-Performance Computer Architecture*, January 2000.

[5] A. Gupta, W.-D. Weber, and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Besed Cache Coherence Schemes. In *International Conference on Parallel Processing*, July 1990.

[6] T. Horel and G. Lauterbach. UltraSPARC-III: Designing Third-Generation 64-Bit Performance. *IEEE Micro*, Volume 19, No. 3, May/June 1999.

[7] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *ACM Annual International Symposium on Computer Architecture*, pages 241-251, June 1997.

[8] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. *In International Conference on Parallel Processing (ICPP'95)*, pages I.1 - I.10, July 1995.

[9] A. Saulsbury and F. Pong and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *23rd Annual International Symposium on Computer Architecture*. May 1996.