Data Integrity Usage Models

Bob Sheffield

Walter Rassbach

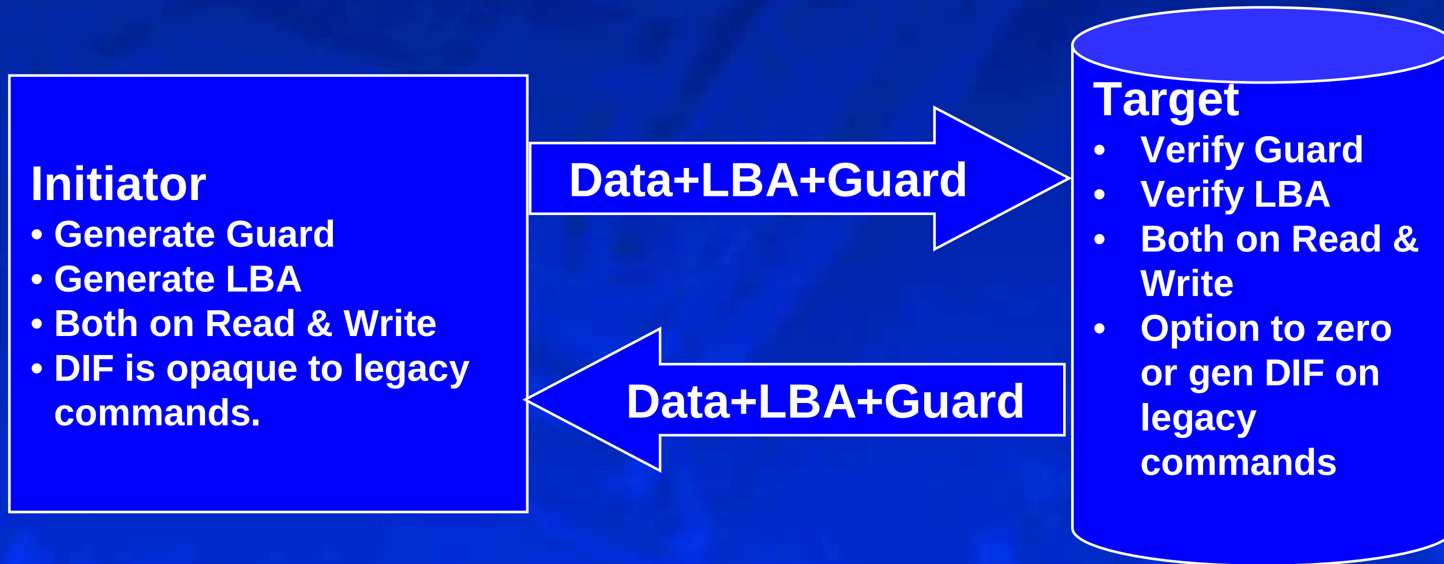
June 30, 2003

DIF Structure Review

Data Block	Excl.	Reference Tag	Meta Tag	Guard
Standard block of data (e.g. 512 bytes)	4-byte incr.	Increments on a per-block basis. 4-bytes	Fixed data – 2-bytes.	2-byte checksum or CRC (Data only)

Key Control	Description	Location
DI Mode Page Supported	Indicates support of Data Integrity Mode Page and DIF media format.	Inquiry byte-5 bit-0
STOR_DIF	Global DIF Enable (also reported in FMT page (03)	DI Mode Page
REF_METHOD	Specifies LBA-Locked or incrementing from base specified in CDB	DI Mode Page
GUARD_METHOD	Specifies checksum or CRC.	DI Mode Page
META_ECHO	Specifies whether initiator or target owns META_TAG field.	DI Mode Page
EXCL_BYTES	Bytes to exclude from guard in main data-block (for stacked DIFs)	DI Mode Page
Pri/Alt/Legacy CK opts	Controls for checking REF, META, & GUARD	DI Mode Page
META TAG MASK (p,a,l)	Ownership Masks for primary, alternate, and legacy META_TAG handling	DI Mode Page
META TAG DEFAULT	Default value assumed for META TAG when not specified in CDB.	DI Mode Page
ZAP_REF & MRK_GRD	Specifies planting ‘0’ or terminal values in REF & Guard fields on legacy writes	DI Mode Page
META_TAG & REF_TAG	Expected values for reads and planted values for writes.	CDB
CHK_OPTN	Specifies to use Primary, Alternate, or Guard Only check controls from Mode Page	CDB
REF_CHK & GUARD_CHK	CMD-by-CMD specification of check options for REF_TAG and Guard	CDB (32-byte)
META_TAG_MASK	CMD-by-CMD specification of META_TAG field split	CDB (32-byte)

Basic Initiator Target



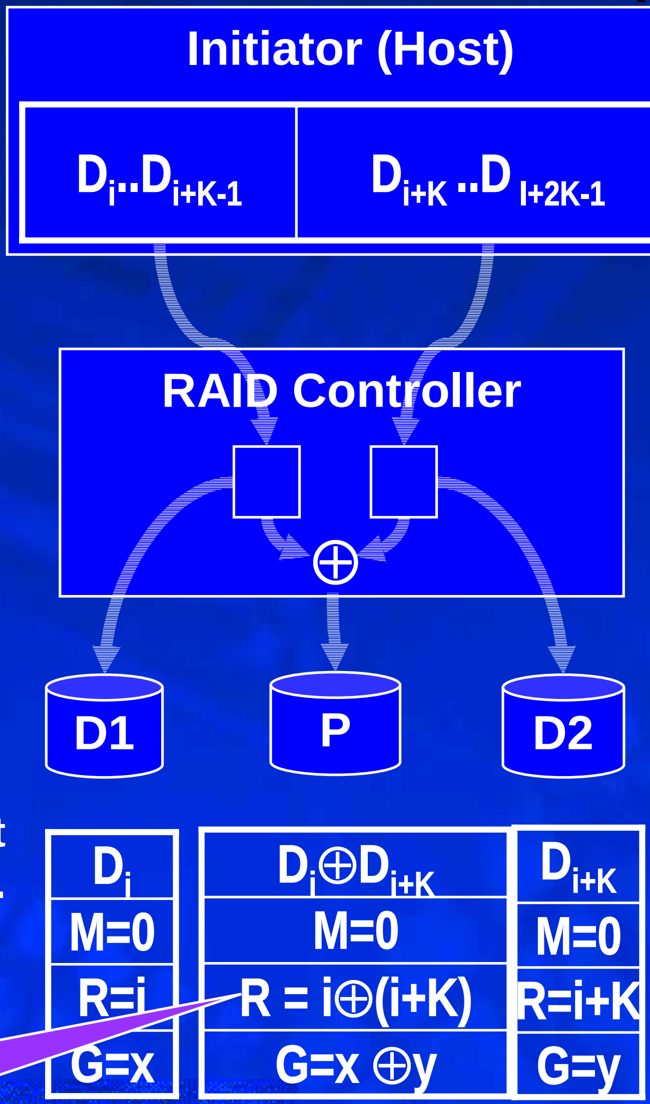
- LBA-Locked Incrementing Tag – used by Initiator to verify disk returns correct block. Works with legacy commands.
- Initiator-generated CRC protects against corruption by HBA.

Basic External RAID-5 with pass-thru DIF

DIF appended / checked by initiator:
- Meta Tag (M),
- Reference Tag (R),
- Guard (G – CRC or Checksum)

Inbound DIF checked by RAID controller on Write. Returned intact on Read.
- REF is LBA-locked for Legacy Read/Write

Avoid consuming extra 8-bytes for back-end DIF by carrying-through host DIF to media unchanged.



Or… for Parity blocks replace w/ R-Tag meaningful to RAID controller.

Reference Tag may be Host LBA-locked (or not if specified in CDB).

DIF for parity block is bit-wise XOR of DIFs from constituent data blocks.

RAID Controller may ‘plant’ unexpected M to flag ‘special’ blocks.

Disks check all DIF fields on data blocks (start-value for Inc. Tag specified in CDB). Guard check only on parity blocks (for CRC).

Basic External RAID-5 with pass-thru DIF

DIF Sub-Field usage (example):

- **Data Blocks:**
  - **REF Tag: Virtual LBA used by the host to access the block.**
  - **Guard: Checksum-based (to obtain full stripe guard characteristics).**
  - **META Tag: Upper byte normally zero (0) indicating a normal data block.**
    Lower byte contains the virtual LUN number
  - **Use Primary Verification controls, checking both REF Tag and Guard value.**
- **Parity Block:**
  - **REF Tag: Lowest virtual LBA in stripe (or equivalent with easy mapping).**
  - **Guard: The XOR of the data block Guard values (stripe protection).**
    - **Note: Guard check on parity is disabled since XOR(G) != G(XOR)**
  - **META Tag: Upper byte is normally 0x80 indicating a normal parity block.**
    Lower byte is virtual LBA of the REF-tag block.
  - **Use Secondary Verification controls, checking REF Tag but not the Guard.**

The upper byte of the META tag is used to indicate the type of block (data or parity) and to hold flags indicating special blocks (e.g., blocks to be explicitly marked as bad) in the lower 7 bits of the upper META Tag byte.

The mask values in the mode page are set to check all bits of the META Tag. A read of a data block will specify 0x00 in the upper byte and the virtual LUN number in the lower byte, while a read of a parity block will specify 0x80 in the upper byte.

Basic External RAID-5 with pass-thru DIF

Types of errors detected* by DIF checking in this model:

- Buffer address calculation errors (e.g., in a scatter/gather list)
- Wild stores to cache or buffer
- Buffer/cache block mis-selections
- Incorrect buffer/cache content (not loaded properly or overwritten)
- During regenerate/rebuild operations: Detection of stripe inconsistencies (e.g., stale data of parity block, possibly due to the RAID 5 "Write hole") – This is a particular property of a checksum guard (and is why it is used).

Many errors are detected only during a Read operation, but checking the DIF during a Write operation will catch a significant portion of these errors. Note that various buffer errors will be detected during destaging operations from cache to the backend and thus their propagation is limited.

This model requires that the host/application must accept a REF Method of 00 (I.e., REF Tags equal to LBA) and that the META Echo control is active (because the RAID controller uses those sub-fields). If the host/application cannot accept those restrictions, then the controller will be forced to "stack" the DIF information.

The model also uses the checksum-based Guard calculation as a means to cover the RAID-5 write-hole.

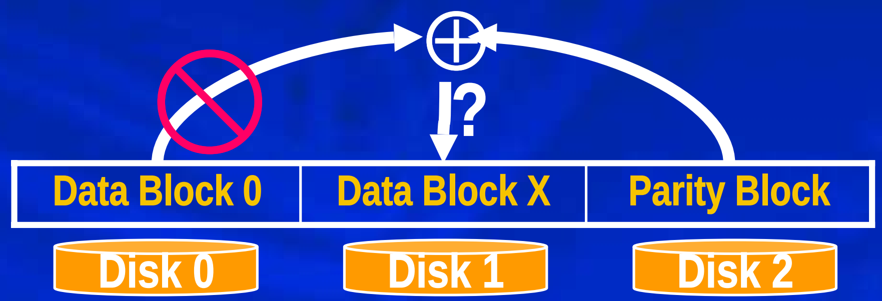
*Note: no error detection method can detect *all* manifestations of a given error type, and there is no "guarantee" that all instances of these errors will be detected, but the majority are detected.

Bad Block Tracking

The disk where Data Block X is mapped to a replacement disk, and data for that disk is being rebuilt from the good disk(s) and parity.

Data-Block zero suffers a grown defect and draws an uncorrectable ECC error.

Issue Read Raw to get as much good data as possible and use that instead.



On subsequent read of Data Block X, the RAID controller needs to:

- Determine the block contains invalid data (trash ECC?)
- Distinguish the block from a defective block (trashing ECC loses distinction – want to bypass reassign/rewrite algorithms – requires VU Write Long)
- Trash Guard – but how do you know it's not a real Guard error?
  - Special Guard value (0 for checksum, special syndrome for CRC-based)
  - But you lose ability to Read Raw Data Block X
- **Use meta-tag that can't be confused with a nominal tag**

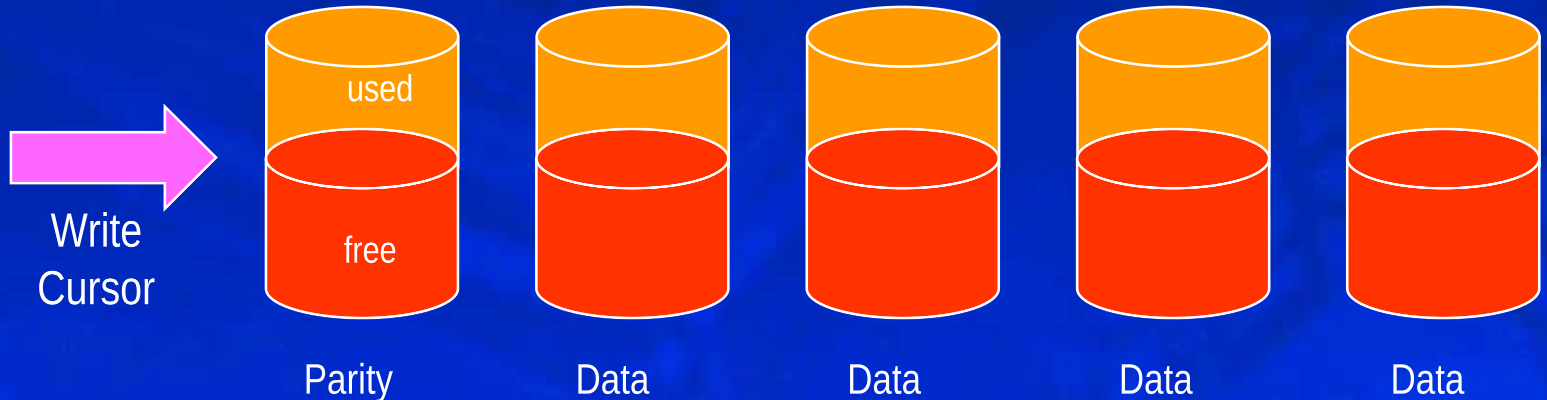
Log-Structured File System model

Basic operation of a log-structured file system:

- Storage is not pre-allocated or assigned (dynamic mapping).
- Write operations allocated and fill free-space blocks rather than overwriting previous version of block.
- Old block images eventually garbage collected and slots added to the free-space lists.
- Read operations to never-written blocks return a default block (or, an error if so configured) – Provide special handling for Write Same.
- Provides automatic “snapshot” capabilities – Older versions of files or data can be obtained at any time until purged/garbage collected.
- Combines well with RAID-style protection – Tends to avoid RAID 5 read-modify-write penalties, generally performing full-stripe physical write operations. Very low seek overheads due to write cursor.

Log-Structured File System model

Because log-structure is generally combined with RAID-style data protection, the model described here assumes such protection.



The stripe at the write-cursor may be only partial, with the META Tag in the parity block indicating how many blocks are included. The next write fills the stripe and (then) updates the parity block, and then starts the next stripe(s). Blocks containing mapping data are interspersed as special data blocks.

Log-Structured File System model

DIF usage for log-structured file system:

- Data blocks:
  - REF Tag: Virtual LBA – NOTE: An implementation might use an index into the mapping tables for sequentiality
  - META Tag: Virtual LUN and control flags
  - Guard: Preferably checksum-based
- Parity blocks:
  - REF Tag: Stripe index (rolling counter)
  - META Tag: Stripe-width and control flags
  - Guard: XOR of data block guard values

Blocks containing mapping information are generally interspersed as special data blocks with META Tags that indicate their usage.

Write operations are generally delayed via write-back cache to allow gathering of contiguous writes before destaging.

Log-Structured File System model

Types of errors (potentially) detected by DIF checking in this model:
- Buffer address calculation errors (e.g., in a scatter/gather list)
- Wild stores to cache or buffer
- Buffer/cache block mis-selections
- Incorrect buffer/cache content (not loaded properly or overwritten)
- Mapping table errors
- During regenerate operations: Detection of stripe inconsistencies

The use of the REF Tag to hold the virtual LBA is very important in this model since the placement of a virtual block is dynamic. The implementation may disable REF Tag checking during operations to the back-end, but REF Tag checking on the host interface is very important to ensure that the correct blocks are transferred.\

The offset from start of LUN is assumed to be consistent across different virtual LUNs to the same device. In other words, all initiators ascribe the same meaning to the REF_TAG even when the access the same target LUN via different virtual LUN IDs.

Object Oriented File System

A controller that presents an "Object Oriented" image to the host (providing an interface that conforms to the pending Object Oriented Device proposals) may well be implemented with back-end storage media that uses the legacy block-oriented architecture. Such an implementation would want to use the DIF in a manner tied to the "object architecture".

One method might be to conceptually combine and then re-partition the META and REF Tag sub-fields as follows:

The exact split between the Object ID and the block index would depend on the exact implementation. Note that objects that exceed the limit of the index field can be assigned multiple Object IDs. Also, note that the REF Tag field (as overlaid) still increments properly through an object.

Object Oriented File System

The controller would accept data from the host and break it into blocks for storage on the back-end. It would attach the object ID and the block-in-object index to each block as it is accepted from the host (during writes) and check those tag values as the data is sent to the host (during reads).

This usage of the DIFs helps to ensure that the controller is properly handling the data and that there are no mapping errors.

Types of errors (potentially) detected by DIF checking in this model:
- Buffer address calculation errors (e.g., in a scatter/gather list)
- Wild stores to cache or buffer
- Buffer/cache block mis-selections
- Incorrect buffer/cache content (not loaded properly or overwritten)
- Mapping table errors

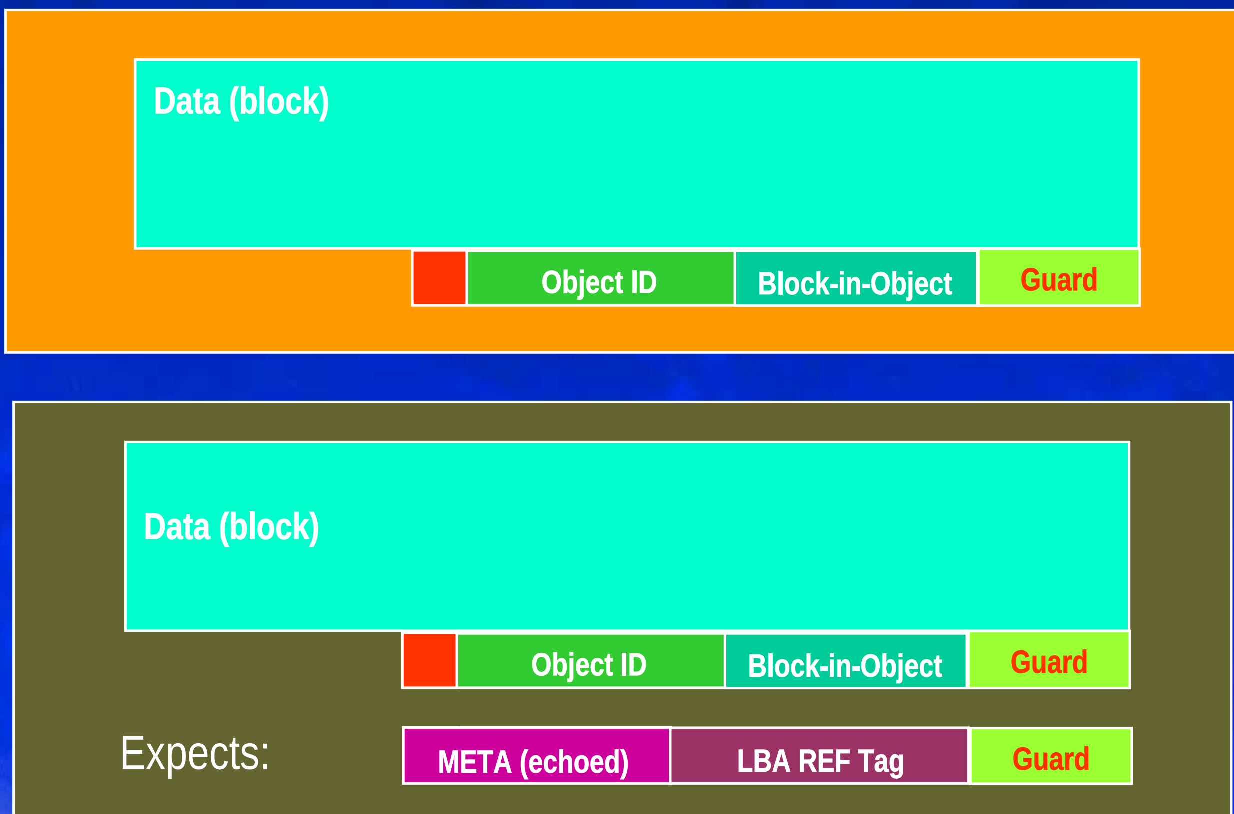
Chained Models

Each of the usage models imposes restrictions on the usage and meaning of the various sub-fields of the DIF. For example, both the simple-RAID and the Log-Structured controllers require that the REF Tag must hold the virtual LBA of the block and are likely to want to impose that restriction on their host if the host is DIF-aware. However, some hosts, e.g., the Object Oriented controller dealing with its back-end, cannot accept such restrictions. The problem is that there is a conflict for the usage of the DIF sub-fields. Similar problems occur if a Log-Structured file system is placed “on top of” (rather than merged with) a RAID implementation (of the type outlined).

The upper level will want to fill in the DIF according to its needs and then transmit it to (and later fetch it back unchanged from) the lower level and have the DIF values checked during the transfer. If the target is really a lower-level controller that wants to impose restrictions on the DIF sub-fields, there is a conflict. The (lower level) controller must either provide a mechanism to deal with this or it will not be compatible/usable in this context.

Chained Models

The situation is illustrated below for an object oriented controller over a simple RAID controller (as described):



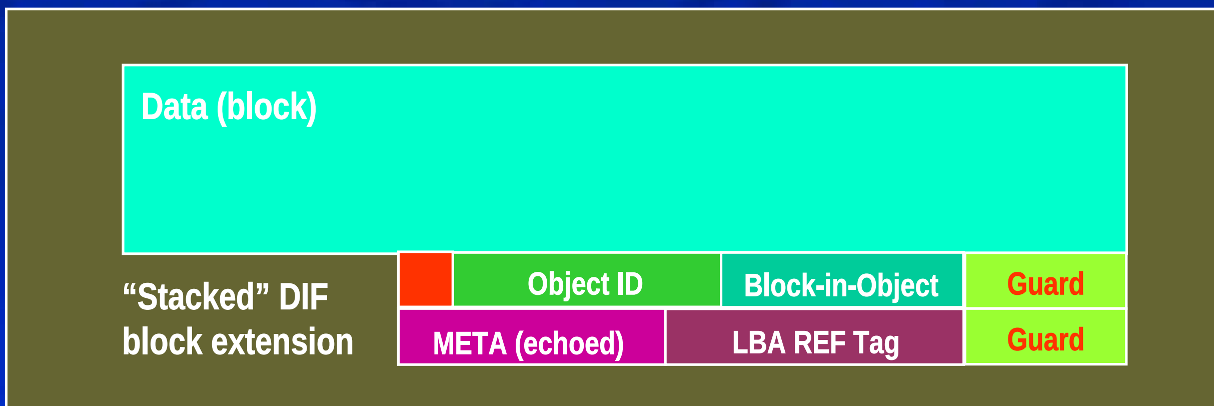
Upper level: Object Oriented

Lower level: RAID, requiring REF as LBA and using META Tag

CONFLICT

Chained Models

The lower level (RAID) controller may provide a solution to this conflict by extending the overall data block with a “stacked” DIF, as follows:



Data block with “stacked” DIF

If the upper-level DIF is excluded from the guard calculation (I.e., the EXCL_Bytes count is increased by 2) and the same guard calculation method is used, the value in the two Guard fields will be identical (since they are calculated against exactly the same data). This makes it much simpler to add the second DIF without having to completely recalculate the guard value.

The lower-level (RAID) controller will append the stacked DIF as blocks are written to it abd strip that DIF when blocks are read.