Appeared in *Proc. of the Fifth International Symposium on High-Performance Computer Architecture* (HPCA'99), Orlando, Florida, January 9-13, 1999.

Permutation Development Data Layout (PDDL) Disk Array Declustering

Thomas J.E. Schwarz°, s.j. Jesse Steinberg* Walter A. Burkhard*

*Gemini Storage Systems Laboratory
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114 USA
{burkhard,steinberg}@cs.ucsd.edu

°Department of Computer Engineering
Santa Clara University
5000 El Camino Real
Santa Clara, CA 95053-0566 USA
schwarz@engr.scu.edu

June 10, 1999

Abstract

Declustered data organizations have been proposed to achieve less-intrusive reconstruction of a failed disk's contents. In previous work, Holland and Gibson identified six desirable properties for ideal layouts. Ideal layouts exist for a very limited family of configurations. The PRIME data layout deviates from the stated ideal only slightly and its run-time performance is very good for light workloads. The DATUM data layout does not meet one of the ideal layout properties but has very good run-time performance for heavy workloads. We present the Permutation Development Data Layout (PDDL) which has excellent run-time performance for both light and heavy workloads. Moreover, PDDL has straightforward space and run-time efficient implementations. Our simulation studies show that the new layouts provide excellent parallel access performance and reduced incremental loads during degraded operation. PDDL can easily accommodate multiple failure tolerant redundancy schemes.

1 Introduction

Many applications require high availability and throughput from cost effective storage subsystems. Disk arrays can offer significant advantages over individual disks improving both throughput and availability of the ensemble. Aggregating multiple disk drives within the array achieves better transfer bandwidth. Storing redundant information provides the mechanism to reconstruct data thereby improving availability. Allocating spare space for the reconstructed data also improves availability. [5, 11, 15]

We propose a novel *data layout,* mapping client data, redundant data and spare space onto the disk array. Data is organized in *stripes* (a.k.a. reliability stripes, reliability groups, parity groups, or clusters). Each stripe contains a fixed number of identically sized *stripe units* and each stripe unit is mapped to contiguous sectors of a disk. To tolerate a single disk failure, all but one stripe unit in a stripe are *data units,* the remaining stripe unit is a *check unit.* The check unit contains the parity of the data units in the stripe. Different stripe units of a stripe are mapped to different disks. The stripe units of an unavailable disk can be reconstructed via the redundancy of the scheme and stored in the allocated *spare space.*

In a *declustered* data organization (originally suggested by Muntz and Liu [13], evaluated by Holland and Gibson [9, 10], and more recently improved upon by Alvarez et al.[1, 2], Merchant and Yu [12], Ng and Mattson [14].

Technical Report CS98-584, CSE, UCSD, April 1998.

Reddy and Bannerjee [16], and Schwabe and Sutherland [17]) each stripe is mapped to k of the n disks in the array (where $k \leq n$), to achieve significant performance improvements during both degraded operation and on-line disk reconstruction. These layouts are based on balanced incomplete block designs (BIBDs) [7, 8] which exist for many array configurations and require the storage of potentially large tables for calculation of the mapping between logical and physical addresses.

Holland and Gibson [9] investigated the *Parity Declustering* scheme in which the mapping is implemented by storing the complete BIBD in a table and using *rotation* to evenly distribute parity. The data layouts discussed this paper, with the exception of PDDL and the Pseudo-Random schemes, are based on BIBDs and can be viewed as special cases of Parity Declustering. We will use the Parity Declustering nomenclature to designate the BIBDs explicitly presented by Holland and Gibson currently at http://www.pdl.cs.cmu.edu/ftp/Declustering/BD_database.tar.Z.

Merchant and Yu [12] propose the *Pseudo-Random* scheme to replace the table lookup with on-demand calculation using pseudo-random permutations to obtain the mapping. Both parity and reconstruction workload are expected to be evenly distributed.

Alvarez, Burkhard and Cristian [1] investigate the DATUM data layout which utilizes the binomial number system (based upon complete block designs) to obtain the mapping; the scheme calculates addresses on-demand. Alvarez, Burkhard, Stockmeyer, and Cristian [2] study the PRIME and RELPR layouts which utilize on-demand calculation to obtain the mapping. These schemes come very close to meeting the layout goals that we discuss below.

Schwabe and Sutherland [17] introduce a slightly relaxed BIBD designs which obtain approximately balanced layouts implemented via on-demand calculation. Reddy and Banerjee [16] as well as Ng and Mattson [14] also present declustering schemes based upon BIBDs.

All these schemes are logically correct; however, the run-time performance varies dramatically depending upon the BIBD. Some of the earlier schemes do not depend upon particular families of BIBDs; the more recent DATUM, PRIME and RELPR data layouts however utilize specific BIBDs. Our motivation is to provide families of data layouts with excellent good run-time performance as well as efficient implementations. We begin by considering desirable properties of our mapping functions. These natural conditions are not easy to meet; Alvarez et al. [2] presents an existential result showing that these properties can be met in very few configurations. Thus we will are forced, in general, to consider schemes that "almost" meet these properties. Our approach is to move beyond BIBDs to *near resolvable designs* [8] which allow sparing as well.

There are six desirable properties of single-failure tolerating declustering layouts, originally identified by Holland and Gibson [9] and discussed in Alvarez et al. [1, 2]:

1. Single failure correcting: No two stripe units of the same stripe are mapped to the same disk.
2. Distributed parity: All disks should have the same number of check units mapped to them.
3. Distributed reconstruction: When any disk fails, its user workload should be evenly distributed over the surviving disks. When renewed, its reconstruction workload should be evenly distributed.
4. Large write optimization: Each stripe contains $k - 1$ contiguous stripe units of client data.

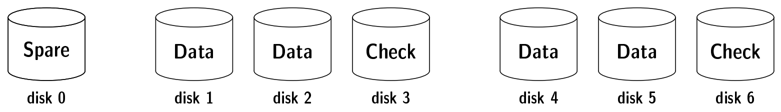

Figure 1: *Virtual RAID Level 4.*

5. Maximal read parallelism: A read of n contiguous data units induces a stripe unit read on each of the n disks.

6. Efficient mapping: The functions that map client logical addresses to array physical addresses are efficiently computable, with low time and space requirements.

These goals were formulated for disk arrays without spare space. A single *distributed spare disk* provides sufficient spare space to store the reconstructed data of a failed disk [11]. The spare space increases data reliability considerably and reduces the workload of individual disks after a failure, once the data on the failed drive have been reconstructed and stored on the spare space. For declustering layouts with n disks, g stripes of width k and a distributed spare disk with $n = gk + 1$, we add the following criteria:

7. Distributed sparing: All disks contain the same number of spare stripe units.

8. Maximal degraded read parallelism: A read of $n - g - 1$ contiguous data units induces a stripe unit read on each of $n - g - 1$ disks during both reconstruction and post-reconstruction operation.

Reconstruction operation occurs prior to writing the failed disk stripe unit to spare space. Post-reconstruction operation occurs when spare space contains the desired stripe unit. Goal #8 is weaker than #5, but if #8 is met, we will be close to meeting #5. Moreover, as we will see later in the paper, having good, if not excellent, response time performance is not generally tied to goal #5 or to #8!

The first five cannot goals can be simultaneously fulfilled for a very limited set of configurations [2]. The contribution of this paper, PDDL satisfies the basic properties #1, #2, #3, #4, #6 and #7 as well as providing good, if not excellent, response time performance. PDDL comes close to satisfying #8. PDDL satisfies #6 very well; a variant of PDDL, presented in the appendix, is a candidate for the fastest possible mapping scheme. There are numerous reasons for considering architectures capable of tolerating multiple concurrent failure [1]. PDDL allows “arbitrary” fixed combinations of check and data blocks.

2 PDDL Data Layout

We describe our approach and demonstrate its simplicity in terms of a small storage server example consisting of seven disks. The storage server user interface will be RAID Level 4 with stripe width three; there are two stripes and one spare disk as shown in Figure 1. A higher level virtual disk user-interface would also be interesting; our approach is applicable in this environment as well as we will demonstrate below. The RAID Level 4 interface uses disks 3 and 6 as parity disks and disk 0 as the spare; the other four disks contain client data.

PDDL declusters the layout by permuting the roles of all seven disks thereby spreading the parity, spare, and client data units throughout the array. The mapping for our storage server example is presented in Figure 2; The left-hand array contains the RAID Level 4 data layout while the right-hand array contains the PDDL declustered layout. The stripe units are labeled with uppercase letters, the spare space with an S, and the check/parity stripe units with P. For example, stripe units A0 and A1 are mapped to disks 1 and 2 while the parity for stripe A is mapped to disk 4. More generally, virtual address (d, l) consists of a disk number d and and a stripe unit number l. In our example, the addresses $(1, l)$, $(2, l)$, and $(3, l)$ always form a stripe for any stripe unit l as do the addresses $(4, l)$, $(5, l)$, and $(6, l)$.

Our mapping is derived from the permutation $(\,0,\ 1,\ 2,\ 4,\ 3,\ 6,\ 5\,)$ which maps the stripe units of the top row of the virtual RAID Level 4 of Figure 2. For example, A1 on disk 2 maps to disk 2 while PA on disk 3 maps to disk 4. The stripe units of the next row are mapped via the permutation $(\,1,\ 2,\ 3,\ 5,\ 4,\ 0,\ 6\,)$. In this row, D1 on disk 5 maps to disk 0 and PD on disk 6 maps to disk 6. The stripe units of the following row are mapped according to the permutation $(\,2,\ 3,\ 4,\ 6,\ 5,\ 1,\ 0\,)$. These permutation have a straightforward association with the permutation for the top row. Specifically, to obtain the permutation for the i^{th} row, we add i mod 7 to the permutation of the top row, coordinate by coordinate. This approach is referred to as *permutation development.* The figure below presents the result of permutation development of $(\,0,\ 1,\ 2,\ 4,\ 3,\ 5,\ 6\,)$; the first column designates spare space, and the fourth and seventh columns designate client data space.

0	1	2	4	3	6	5
1	2	3	5	4	0	6
2	3	4	6	5	1	0
3	4	5	0	6	2	1
4	5	6	1	0	3	2
5	6	0	2	1	4	3
6	0	1	3	2	5	4

The client data is stored within the disk array by repeating the seven row layout pattern. The mapping function for our seven disk storage server example can be easily implemented as follows:

```
int permutation[ ] = { 0, 1, 2, 4, 3, 6, 5 } ;

int virtual2physical( int disk , int offset )
{
       return ( ( permutation[disk] + offset ) % 7 ) ;
}
```

Our permutation development mapping scheme evenly distributes spare and check space. Since each column contains each of the seven disks names once, we have each disk containing $1/7^{th}$ of the total spare space, $2/7^{ths}$ of the total parity space and $4/7^{ths}$ of the data space. Thus, we meet goals #1, #2, #4 and #7 of the desirable properities list. These data layouts do not meet the maximal read parallelism goal #5 but we do have a very efficient mapping if not the most efficient thereby meeting goal #6. We return to the need to meet goal #5 subsequently.

The distributed reconstruction goal #3 is examined next. Suppose disk 0 fails. Within the left stripe of the previous figure, row 3 indicates that disks 4 and 5 must be accessed to reconstruct the parity unit of the stripe that was on disk 0 and which will be stored on the spare space of disk 3. Similarly, row 5 indicates that disks 2 and 6 must be accessed to reconstruct the client data that was on disk 0; the reconstructed value is stored on disk 5 spare space. Finally, we access disks 1 and 3 according to row 6 to reconstruct client data that will be stored on disk 6 spare space. Each of the surviving disks are accessed once for this process and disks 3,5 and 6 are written once. For the right stripe, similar analysis shows that each of the surviving disks is accessed once to reconstruct data and disks 1, 2, and 4 are

disk 0	disk 1	disk 2	disk 3	disk 4	disk 5	disk 6
S	A0	A1	PA	B0	B1	PB
S	C0	C1	PC	D0	D1	PD
S	E0	E1	PE	F0	F1	PF
S	G0	G1	PG	H0	H1	PH
S	I0	I1	PI	J0	J1	PJ
S	K0	K1	PK	L0	L1	PL
S	M0	M1	PM	N0	N1	PN

Virtual RAID Level 4

disk 0	disk 1	disk 2	disk 3	disk 4	disk 5	disk 6
S	A0	A1	B0	PA	PB	B1
D1	S	C0	C1	D0	PC	PD
PF	F1	S	E0	E1	F0	PE
PG	PH	H1	S	G0	G1	H0
J0	PI	PJ	J1	S	I0	I1
K1	L0	PK	PL	L1	S	K0
M0	M1	N0	PM	PN	N1	S

Physical Disk Array

Figure 2: *PDDL Mapping Example.*

each written once. The permutation development structure of this mapping gives exactly the same evenly distributed reconstruction workload results as in the previous paragraph for any other failed disk as well. Thus, the reconstruction workload is evenly divided over the surviving disks for any failed disk.

There is another point regarding our example – the selection of permutation $(0, 1, 2, 4, 3, 6, 5)$. We refer to this permutation as the *base permutation* as it provided the underpinnings of our mapping. *Any* permutation, utilized as a base permutation, will meet goals #1, #2, #4, #6 and #7 because of the permutation development structure. Moreover, our base permutation meets goal #3 as well. However, the distributed reconstruction goal #3 is not always met. For example, if we use the permutation $(0, 1, 2, 3, 4, 5, 6)$ as our base permutation, the reconstruction workload is spread over only four disks instead of all six surviving disks. Two of the four disks will be reading two stripe units instead of one increasing their workload by 50%. This permutation does not meet goal #3. These issues are explored in more detail in the next section; we present some (known) techniques for selecting base permutations, referred to as *satisfactory base permutations*, that evenly distribute the reconstruction workload over all surviving disks.

Another slight generalization of our approach utilizes more than one base permutation in situations where a solitary satisfactory permutation cannot be found. For example, if n is ten and k is three, we can utilize a pair of base permutations, neither of which satisfy goal #3, to obtain an evenly distributed reconstruction workload. Here are the base permutations — $(0, 1, 2, 8, 3, 5, 7, 4, 6, 9)$ and $(0, 1, 2, 4, 3, 7, 8, 5, 6, 9)$. Let's assume disk 0 has failed. The reconstruction workload tally for the first base permutation is

disk	1	2	3	4	5	6	7	8	9
stripe units	1	3	2	2	2	2	2	3	1

and for the second base permutation

disk	1	2	3	4	5	6	7	8	9
stripe units	3	1	2	2	2	2	2	1	3

These two base permutations provide a 20 row layout pattern with reconstruction workload of four reads and two writes for each of the surviving disks per layout pattern.

Number of Stripes	*Stripe Width*					
	5	6	7	8	9	10
1	1	1	1	1	1	1
2	1	1	2	1	1	?
3	1	1	1'	2	2	1
4	1	1	1	1'	1	1
5	1	1	1'	1	3	2
6	1	1	1	3	6	1
7	1	1	2	5	?	1
8	1	2	4	?	1	?
9	2	2	5	1	?	?
10	1	1	1	?	?	1

Table 1: *Satisfactory PDDL base permutations; an apostrophe denotes a solution for a non-prime number of disks obtained or compiled by Furino[8].*

3 PDDL Base Permutations

Satisfactory base permutations are the fundamental component of the PDDL scheme; furthermore these permutations remain constant. Our goal is to efficiently obtain satisfactory base permutations (or groups of base permutations) given the stripe size k and the number of stripes g where the number of disks n is $g\,k + 1$.

If n is a prime, the well-known construction of Bose [3], which we describe here, always yields a solitary satisfactory base permutation. This construction creates a *near resolvable design* which is a well-studied combinatorial object. The Bose construction also works when n is a power of a prime but the results generally are less attractive for our purposes.

1. Determine a primitive element ω; ω is a primitive if each non-zero element can be expressed as a power of ω modulo n.

2. Distribute all non-zero elements in round-robin fashion into g subsets $B_1, B_2, \ldots, B_g$ as follows.

$$\begin{aligned} B_1 &= \{\, \omega^0,\, \omega^g,\, \ldots\, ,\, \omega^{g(k-1)} \,\} \\ B_2 &= \{\, \omega^1,\, \omega^{g+1},\, \ldots\, ,\, \omega^{g(k-1)+1} \,\} \\ &\ddots \\ B_g &= \{\, \omega^{g-1},\, \omega^{2g-1},\, \ldots\, ,\, \omega^{g\,k-1} \,\} \end{aligned}$$

The base permutation is

$$(\, 0,\, \omega^0,\, \omega^g,\, \ldots\, ,\, \omega^{g(k-1)},\, \omega^1,\, \omega^{g+1},\, \ldots\, ,\, \omega^{g(k-1)+1},\, \ldots\, ,\omega^{g-1},\, \omega^{2g-1},\, \ldots\, ,\, \omega^{g\,k-1} \,)\,.$$

As an example of the algorithm, we construct the permutation for our seven disk storage server. We have $n = 7$ and $g = 2$. 3 is a primitive element since 3^0 is 1, 3^1 is 3, 3^2 is 2, 3^3 is 6, 3^4 is 4, and 3^5 is 5; these powers are evaluated modulo 7. Then $B_1 = \{\, 1, 2, 4 \,\}$ and $B_2 = \{\, 3,\, 6,\, 5 \,\}$ and our base permutation is $(\, 0,\, 1,\, 2,\, 4,\, 3,\, 6,\, 5 \,)$.

We are left to consider configurations in which n is not a prime. When n is a power of a prime, the Bose construction obtains a satisfactory base permutation; however, the addition operation utilized in our mapping function must be

Workload	
Access sizes:	8,24,48,72,96,120,144,168,192,216,240,288,336 KB
Concurrency:	1,2,4,8,10,15,20,25 clients
Alignment:	8 KB (stripe unit boundary)
Random accesses uniformly distributed over all data	
Array	
Stripe unit:	8 KB
Data layouts:	PRIME
	Parity Declustering
	Left-symmetric RAID-5
	PDDL
	DATUM
Number of disks, n:	13
Stripe width, k:	Prime, Parity Declustering, PDDL,
	DATUM: 4 stripe units
	RAID-5: 13 stripe units
Disk (HP 2247)	
Capacity:	1.03 GB
Geometry:	1981 cylinders, 13 heads, 8 zones
Average seek time:	10 ms.
Rotational speed:	5400 RPM (11.12 ms./rev.)
Head scheduling:	SSTF on 20-request queue

Table 2: *Simulation parameters.*

addition within the underlying finite field $GF(n)$. For prime n, the addition operation is addition modulo n and for n a power of 2, the addition operation is bitwise exclusive-or which is available in most hardware environments. We continue this discussion in the Appendix.

To further our knowledge of satisfactory base permutations, we have implemented computer searches to locate both solitary as well as multiple base permutations. Using simple hill-climbing from random starting points, our program locates permutations which are satisfactory or almost satisfactory. If it cannot find a satisfactory permutation, it combines almost satisfactory permutation into small groups of base permutations that obtain evenly distributed reconstruction workload. For n up to 60 and small stripe sizes, we have been fairly successful in locating satisfactory base permutations for which modular addition suffices. Table 1 contains our current results.

4 Performance

We compare the performance of our PDDL data layouts with DATUM[1], PRIME [2], Parity Declustering [9], and RAID-5 [15]. Parity Declustering was chosen as the initial and typical representation of BIBD-based layouts; RAID-5 satisfies maximal parallelism optimally, even though it is not a declustered layout; PRIME almost satisfies maximal parallelism optimally with a deviation of one from optimal; and DATUM provides excellent performance under heavy workloads.

Our experiments were run on RAIDframe [6], a disk array architecture test-bed for run-time performance evaluation. RAIDframe and its predecessor Raidsim have been used in several experiments [1, 2, 6, 9]. Table 2 shows the

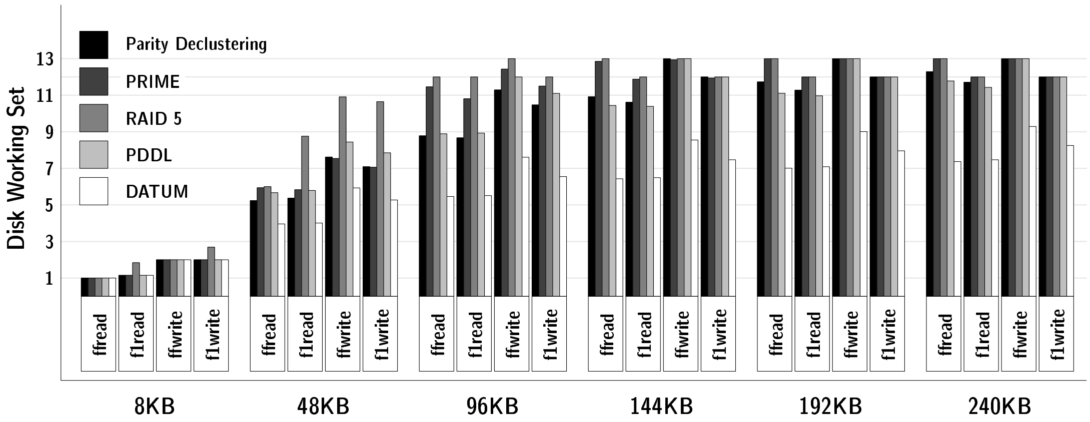

Figure 3: *Disk Working Set Sizes.* Calculated by averaging the working set sizes for logical accesses for every possible offset in the array. The prefix “ff” denotes failure-free operation, while the prefix “f1” designates single disk failure. For PDDL, f1 designates the reconstruction mode.

parameters for the array and disk simulator. The workload serviced by the array consists of interleaved streams of fixed-size *logical accesses* of the same type (read or write), originated by a fixed number of simulated clients. Workloads are synthetic: each client generates an independent logical access starting at a random location with uniform distribution, the client then blocks until the array services the logical access, and immediately repeats the cycle. Logical accesses span an integer number of stripe units, and are aligned to a stripe unit's boundary. Traces or synthetic workloads with a more realistic access mix would be a better predictor of the performance of the arrays in a real situation. Our goal is to analyze the strengths and weaknesses of the architectures with respect to each other; several subtle interactions and tradeoffs would have been considerably more difficult to understand with less homogeneous workloads. The wide range of access sizes is considered to observe the (possibly extreme) effects of the various data layouts. A very interesting question we leave open here is the issue of the optimal stripe unit size; this has been considered for Level 5 RAID [4].

Logical accesses are translated by the array controller into sets of *physical accesses* on the individual disks. Our simulated arrays always have 13 disks. Usable storage capacity varies according to the layout: RAID-5 uses 7.7% of the disks for parity, PRIME, DATUM and Parity Declustering have a parity overhead of 25%, and PDDL has a parity overhead of 23.1% plus spare overhead of 7.8% in our configuration. RAID-5 and Parity Declustering can tolerate one failure by definition; the configuration of PRIME, DATUM and PDDL used in our experiments also tolerate single failures. Each disk controller performs dynamic request reordering following the shortest-seek-time-first (SSTF) policy to choose the next request to be serviced from the disk's queue.

Our principal performance metric is the *access response time*: the average time elapsed from the moment a client requests a logical access, to the moment the array completes the access. Given that both the number of clients and the size of logical accesses are fixed during each experiment, throughput can be easily calculated from the response times reported here: Throughput = Request_Size * Number_of_Clients / Response_Time. Experiments run until the measured access response time is within 2% of the true average with 95% confidence.

For logical access l, we define its *disk working set size* as the number of disks that perform at least one physical access to process l. Figure 3 shows the disk working set size for the data layouts, for various access sizes: 1, 6, 12, 18, 24, and 30 consecutive stripe units. For each access size we consider reads and writes separately, and for each access type we consider both failure-free and degraded modes (when one disk has crashed and has not been reconstructed). RAID-5 satisfies the maximal parallelism property optimally. PRIME has a maximum deviation of one from the optimal. Neither Parity Declustering nor DATUM satisfy this property. This can be observed in Figure 3 for failure-free reads when no parity unit will be accessed. The disk working set size, designated DWS, are related as follows for our access sizes up to 120KB:

$$DWS(\text{DATUM}) \leq DWS(\text{Parity Declustering}) \leq DWS(\text{PDDL}) \leq DWS(\text{PRIME}) \leq DWS(\text{RAID-5})$$

And for our access sizes larger than 120KB:

$$DWS(\text{DATUM}) \leq DWS(\text{PDDL}) \leq DWS(\text{Parity Declustering}) \leq DWS(\text{PRIME}) \leq DWS(\text{RAID-5}).$$

The relative sizes for Parity Declustering and PDDL switch in these orderings. RAID-5 peaks out at the maximum possible values (13 disks for failure-free mode, 12 disks for degraded mode) for smaller access sizes than the other layouts. In fact, Parity Declustering, DATUM and PDDL do not reach these maximum values for any read size in the figure. Since the average number of physical accesses per logical access is the same for any declustered layout with the same values of n and k, layouts with smaller working sets map more accesses to each disk. Accesses mapped to the same disk are serviced sequentially in some order; therefore, larger working sets imply fewer serialized operations.

PDDL contains distributed sparing; we could provide reconstruction and post-reconstruction response time performances. The other data layout schemes do not contain sparing and we provide only reconstruction response time performance for all in Figures 6 and 9. We present PDDL post-reconstruction read response times in Figure 18 in the Appendix. For accesses much larger than the stripe unit size, the post-reconstruction and reconstruction reponse times are practically the same since we do essentially the same work in either case. For approximately stripe unit sized accesses, the post-reconstruction response time is much better than the reconstruction response time. The post-reconstruction response time performance, even for stripe unit sized accesses, is worse than the fault free read response times since there is one less operational disk. We note that selecting the stripe unit equal to the access size can greatly improve the read response time in post-reconstruction operations.

4.1 Reads

Figure 5 shows the response times for failure-free read accesses; we present a complete set of figures and tables within the Appendix for the reviewers. Since the conference proceedings will limit our page count, ultimately we confine the Appendix to our report [18]. In the 8KB case, performance is very similar for these layouts. However, we observe the following behavior for larger accesses: PDDL is faster than both Parity Declustering and DATUM when the array is lightly loaded but as the load increases, the response time curves cross and PDDL becomes more efficient and ultimately its run-time is better than all but DATUM.

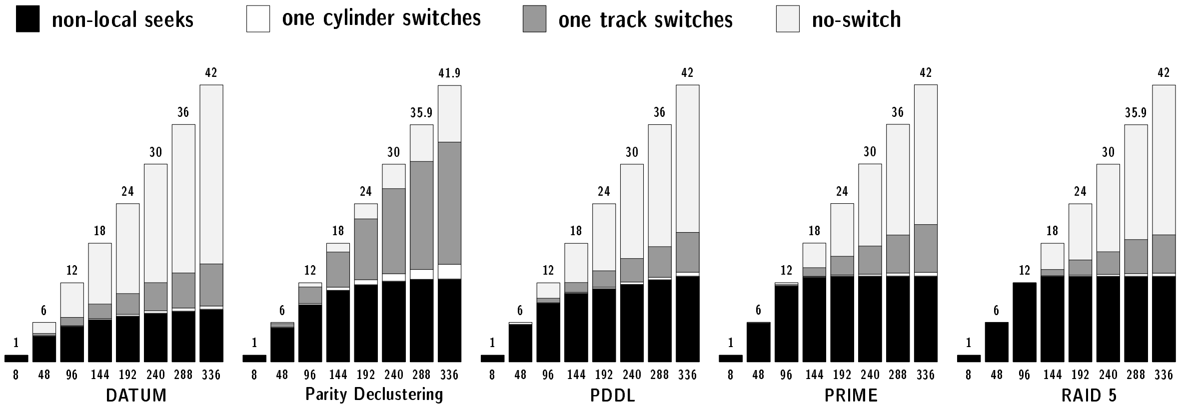

Figure 4: *Fault free read; seek and no-switch counts*

This effect occurs for all failure-free reads smaller than 240KB. To understand it, we consider seek overheads. Figure 4 shows the average number of seeks and no-switch operations (defined below) for each logical access size from 8KB through 336KB measured during our simulations. The number of seek and no-switch operations is almost independent of the workload. Each column contains four components; the black designates non-local seeks which are defined below. The other three designate local operations; the white label designates seeks resulting in a cylinder switch, the dark grey label designates seeks resulting in a head switch within a cylinder, and the light grey label designates operations incurring only rotational delay rather than a cylinder or head switch. We classify operations as either "local" or "non-local." Local operations occur between successive operations caused by the same logical access; the remainder are non-local. Local operations involving seeks are further classified as a cylinder switch or a track switch; we use the name "no-switch" to designate the remaining local operations. In our simulations, the cylinder switch service time is 2.9 ms., the track switch service time 0.8 ms., and the no-switch service time is less than 5.6 ms. The response times for the three varieties of operations are essentially the same for each of the data layouts; however, their response times differ considerably among the layouts.

The non-local seeks counts obtained in our experiments and presented in Figure 4 and the working set sizes from Figure 3 are equal; moreover, they are determined independently. Since the shortest seek-time first head scheduling policy is utilized, a disk will seldom alternate between logical accesses. Most often each operation of a logical access will be completed before beginning the next access. Therefore, layouts with the larger working sets (PRIME, RAID 5, PDDL, Parity Declustering) incur more non-local seeks during the execution.

For 8KB stripe units, the data transfer time is negligible compared to the head positioning time. Layouts with smaller numbers of seeks will achieve smaller response times. There is, however, an important addition to this rule: when the array is experiencing a light workload, the disks are relatively idle and can respond in a short interval. In this situation, larger working sets imply better performance because the array takes advantage of the available bandwidth by performing more parallel accesses. Similarly, when the array is experiencing a heavy workload, the disks are all busy and the access will be queued at each disk. In this situation, smaller disk working sets imply better performance

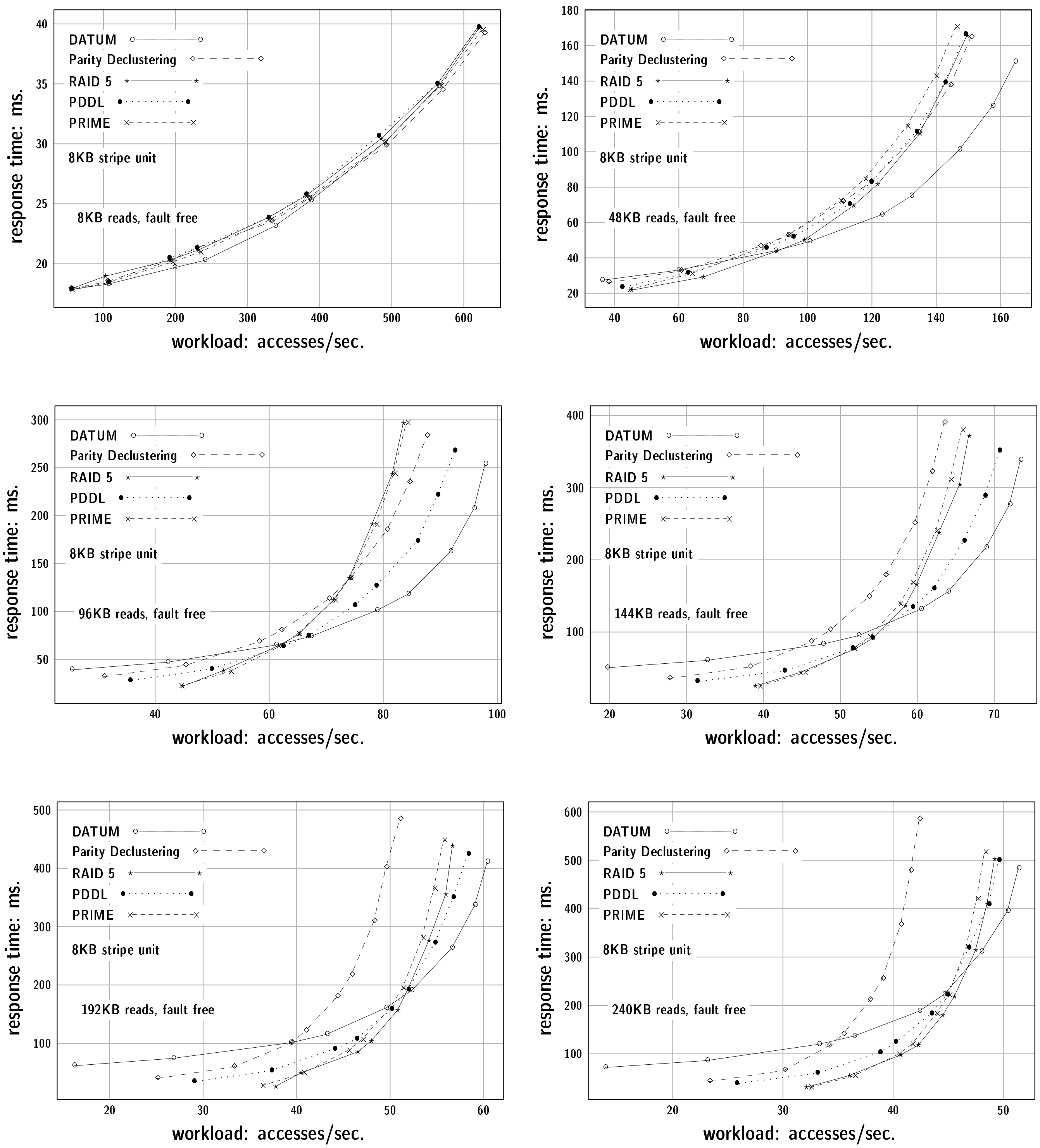

Figure 5: *Read response times: Failure-free mode.*

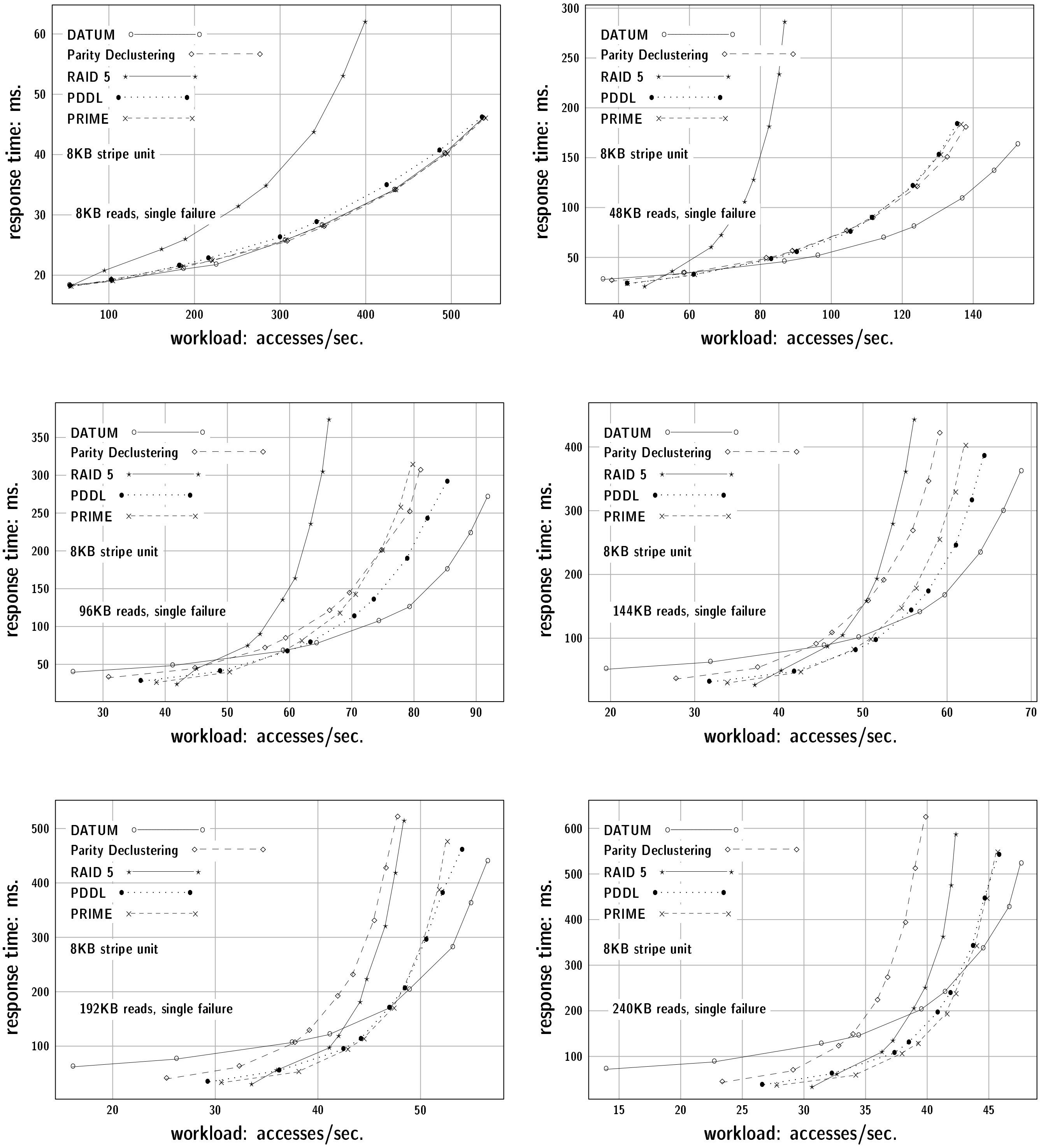

Figure 6: *Read response times: Single failure mode.*

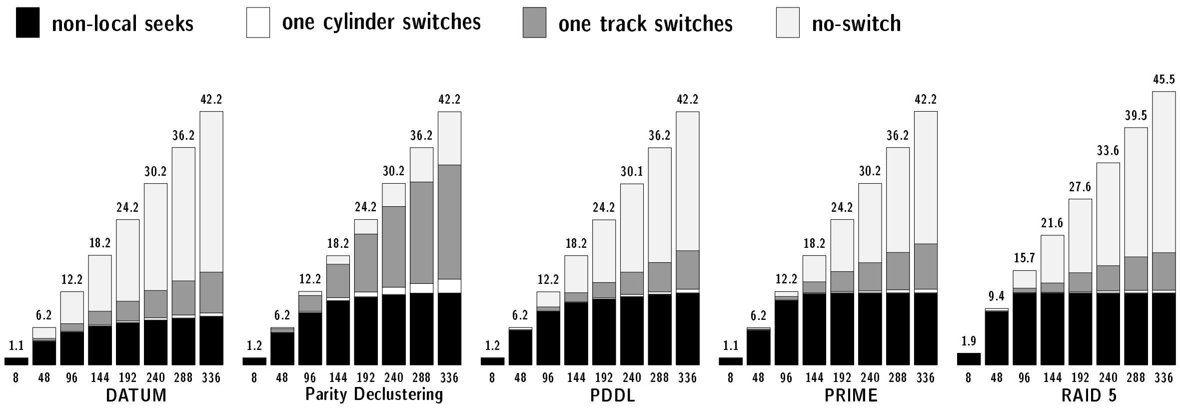

Figure 7: *Degraded read; seek and no-switch counts*

because the queuing time dominates the access run-time.

The run-time performances for the data layouts, are analyzed by considering the number and response times of the various local operations as well as the disk working set sizes. In Figure 4, all non-black portions of the columns represent serialized behavior.

DATUM has the smallest disk working set of these data layouts; accordingly, its relative run-time performance is poor for light to moderate workloads but it improves dramatically becoming the very best for heavy workloads. The local operations have little impact on this since their response time is due primarily to inexpensive no-switch operations.

PDDL has a relatively small disk working set; its run-time performance is better than DATUM and Parity Declustering for light to moderate workloads. The local operations, primarily no-switches, have more impact as the access size increases; the run-time performance is second best for heavy workloads.

PRIME has a large disk working set; its run-time performance is among the very best for light to moderate workloads. As the workload increases, both the local operations and the very large disk working set dramatically impact the run-time performance; its run-time is second worst for heavy workloads.

The RAID 5 layout is better than PDDL for light to moderate workloads due to its large disk working set. As the workload increases, the impact of the expensive local operations is small. For large sizes, its run-time performance is better than all but PDDL and DATUM.

Finally, Parity Declustering is better than DATUM for light to moderate workloads; the relative sizes of the disk working sets for Parity Declustering and PDDL switch at 120KB access size as mentioned previously. However, the run-time for Parity Declustering under light to moderate workloads is worse than for PDDL. Parity declustering has more costly local operations than PDDL. For heavy workloads, its large disk working set and relatively expensive local operations give rise to the worst run-time performance of these data layouts.

Similar arguments can be made for the run-time performance of degraded reads given in Figure 6. The seek and no-switch tallies measured in our experiments and presented in Figure 7 are similar to those for fault-free reads. The primary differences in the relative performances are quantitative. However, RAID-5's run-time performance degrades significantly; this phenomenon is, in fact, the rationale for declustering. Within RAID-5, the workload on the surviving disks doubles during degraded read accesses.

4.2 Writes

Figure 8 shows the response times for failure-free writes. Again, response times are very similar for all the layouts in the 8KB case. For all larger accesses, PRIME, DATUM and PDDL show better performance than Parity Declustering — and the difference increases with access size. The seek and no-switch tally distributions are very similar to those of fault free reads.

RAID-5 has much higher response times than the declustering layouts for 48KB accesses. This is because, even though all these layouts satisfy the large write optimization, the stripe size is 13 for RAID-5, compared with 4 for the other layouts. Therefore, writes to a whole stripe will occur much more often for the declustered layouts than for RAID-5. In fact, RAID-5 is implementing all its writes as "small writes" in the 48KB case—that is, reading the old values of data and parity, using them to compute the new parity, and writing the new data and parity. It is easy to see that this requires at least twice as many physical accesses than full-stripe writes. This effect is less pronounced for larger access sizes, but PRIME, DATUM and PDDL have a substantial performance advantage in our experiments especially as the access size increases.

For degraded writes, the response times, presented in Figure 9, of the declustered layouts are slightly better than in the failure-free case (by a very small margin). This is a well-known phenomenon: for degraded write accesses, the array actually does *less* work in many cases when performing large writes, because the failed disk cannot be written. The RAID-5 response times degrade with respect to failure-free mode, more significantly for smaller access sizes. The relative disk working set sizes as well as the local operation run-time overheads give rise to the response times. For 8KB and 48KB accesses, all logical writes span less than half a stripe, so RAID-5 would ideally implement them as small writes. However, when one of the disks containing modified data has crashed, every logical write must be implemented as a "large write"—that is, reading the stripe units that will not change instead of the ones that will. Therefore, while large writes never occur for RAID-5 in failure-free mode, they are quite common in degraded mode. Moreover, since no logical write involves more than half the data units of the stripe, implementing it as a large write results in more physical reads. The average number of physical accesses per logical write is therefore higher in degraded mode. This effect is less pronounced for larger access sizes, when some large writes do occur even in failure-free mode.

4.3 Time and Space Performance

The response time performance of Figures 5, 6, 8, and 9 is a principal component of the effectiveness of a disk array data layout design. We consider several other performance metrics here. The first is the size of memory required to implement the mapping function, the second is the size of the layout pattern which we refer to as the *period* of the layout, the third concerns sparing, and finally, the fourth (slightly redundant) is the efficiency of the mapping

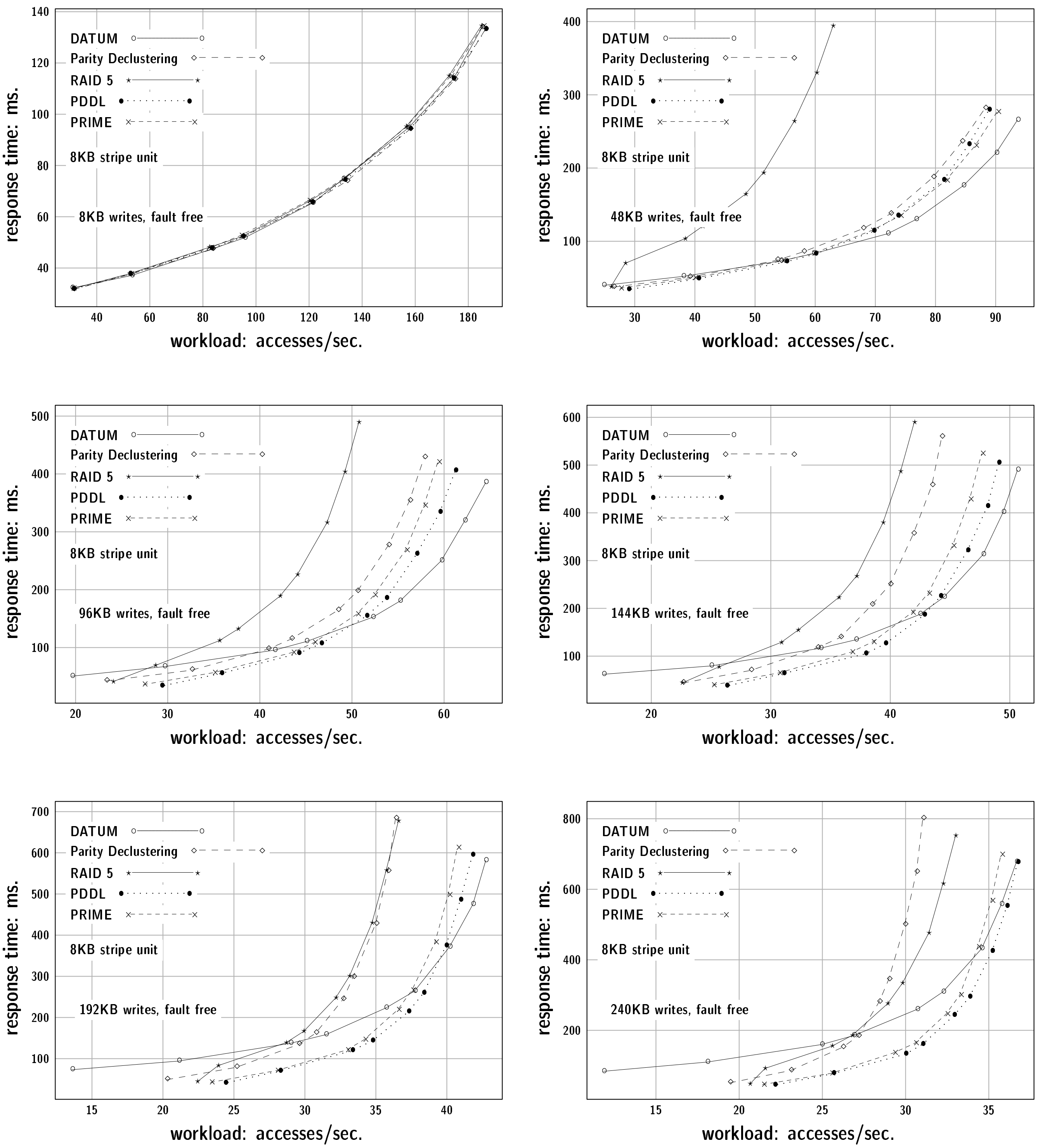

Figure 8: *Write response times: Failure-free mode.*

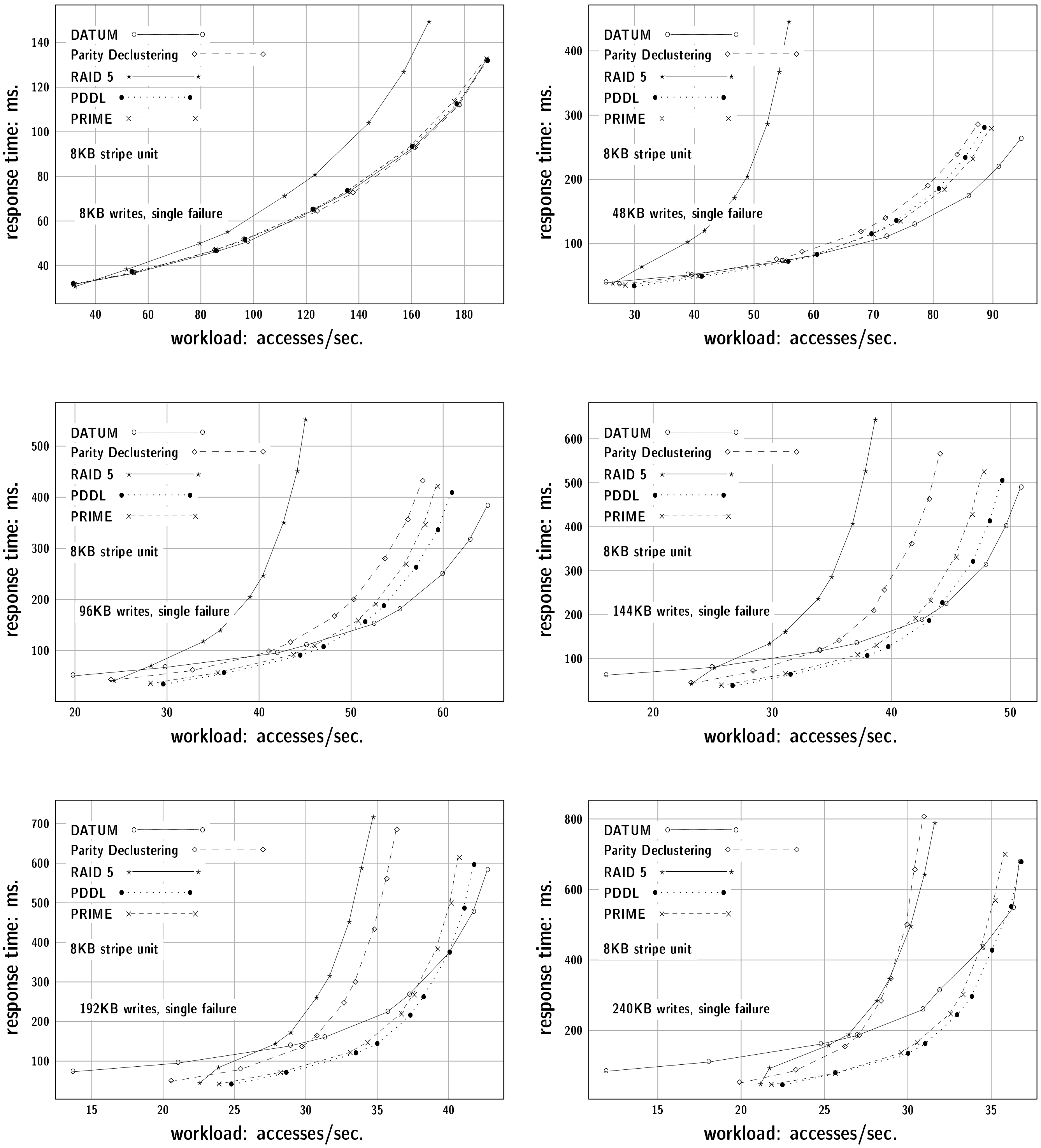

Figure 9: *Write response times: Single failure mode*

Scheme	Table Size	Translation Time	Sparing	Period
Parity Declustering	$\lambda \frac{n(n-1)}{k-1}$	table lookup & parity rotation	no	$\lambda \frac{k(n-1)}{k-1}$
Pseudo-Random	$\log(n)$ $+\log(D)$	Thorpe's Shuffle & table lookup	optional	not applicable, expected values only
DATUM	0	few arithmetic operations	no	$k\binom{n-1}{k-1}$
PRIME	0	few arithmetic operations	no	nk
PDDL	pn	very few arithmetic operations & vector lookup	yes	pn

Table 3: *Comparison of PDDL with other declustering schemes.*

function evaluation. Table 3 contains a comparison of PDDL, with p base permutations, and the Parity Declustering, Pseudo-random, DATUM, and PRIME schemes with respect to these metrics.

5 Conclusions

We have presented PDDL a novel declustering scheme that provides excellent response time behavior. PDDL does meet our goals #1, #2, #3, #4, #6, and #7, but PDDL does not satisfy the maximal read parallelism goal #5. However, PDDL does meet goal #8 for *super stripes* which are row-aligned $n - g - 1$ contiguous stripe units but not for general, contiguous $n - g - 1$ stripe unit accesses. Meeting goal #8 for super stripes makes PDDL an extremely attractive approach for very large data accesses.

PDDL's response time is at least comparable and often better than to all but DATUM for heavy workloads. Whether a storage server would be routinely pushed to this level of operation is subject to debate and the answer is probably no. However, in situations where such levels do occasionally arise, PDDL expeditiously carries out its tasks. For light to moderate workloads, PDDL has among the very best response times especially for write intensive workloads; for read intensive workloads, it is also very competitive. There is always the situation of designing a server in which the "typical" access size is determined but then some client will want to make larger accesses. PDDL very efficiently meets these additional demands as well. PDDL makes very small implementation demands regarding mapping response-time complexity and mapping space.

Given the excellent performance of PDDL, the issue of whether the stated goals we listed in the introduction are correct should be considered. In particular our goals #5 and #8 seem to have little direct correlation with response time performance except possibly for light workloads. Possibly some slightly different goals could be stated – one such goal might include the disk working set size as well as seek variety counts. We have considered the issue but have only preliminary results now.

PDDL's range of application is wide. The number of disks must be prime or as given within Table 1; the number is a multiple of the stripe width plus one for the spare disk. While there is no generic way to find a single base permutation or a group of base permutations, our experiments and analysis indicate that this does not pose an obstacle for a large range of disk array configurations. Very recently, we have discovered a method, referred to as *wrapping*, to combine PDDL and DATUM thereby obtaining the benefits of both as well as expanding the range of applicability. For example, to create a data layout for 30 disks with stripe width seven, we first create a DATUM layout with stripe width 29. Then for each of the 30 rows of the DATUM layout, we use the PDDL data layout with four stripes each of width seven plus a spare. The resulting data layout meets our goals #1, #2, #3, #4, #6 as well as #7. This topic will be presented in a future paper.

In comparison with the other schemes, PDDL has a much less complex translation algorithm. PDDL uses only p base permutations and its storage requirement is very moderate and proportional to p. PRIME and DATUM however do not use any tables. PDDL is based on sparing. Since bandwidth is more important than storage capacity and since the provision of a spare is one of the most effective way to increase mean time to data loss, distributed sparing [11] is a sure win. PDDL can even be altered to have more than one spare disk distributed in the disk array. PDDL can be adjusted to schemes using more than one check block per stripe. Wherever PDDL is possible, it is an attractive choice over the competition.

References

[1] G.A. Alvarez, W.A. Burkhard, F. Cristian: "Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering", *Proceedings of the 24th Annual ACM/IEEE International Symposium on Computer Architecture,* p.62-72, June 1997.

[2] G.A. Alvarez, W.A. Burkhard, L.J. Stockmeyer, F. Cristian: "Declustered Disk Array Architectures with Optimal and Near-optimal Parallelism," *Proceedings of the 25th Annual ACM/IEEE International Symposium on Computer Architecture,* June 1998.

[3] R.E. Bose: "On the Construction of Balanced Incomplete Block Designs", *Annals of Eugenics,* vol. 9, p. 353-399, 1939.

[4] P.M. Chen, E.K. Lee: "Striping in a RAID Level 5 Disk Array," *Proceedings, ACM SIGMETRICS '95,* pp. 136-145, May 1995.

[5] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, D.A. Patterson: "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Surveys,* vol. 26, no. 2, pp. 145-185, June 1994.

[6] W. Courtright, G. Gibson, M. Holland, J. Zelenka: 'A structured approach to redundant disk array implementation', *Proceedings of the International Symposium on Performance and Dependability*, pp. 11-20, 1996.

[7] **The CRC Handbook of Combinatorial Designs;** edited by Charles J. Colbourn and Jeffrey H. Dinitz, CRC Press, Boca Raton, 1995.

[8] Steven Furino, Ying Maio, Jianxing Yin: **Frames and Resolvable Designs: Uses, Constructions, and Existence;** CRC Press, Boca Raton, 1996.

[9] M. Holland, G.A. Gibson: "Parity Declustering for Continuous Operation in Redundant Disk Arrays", *Proceedings ASPLOS-V,* pp. 23-35, Sept. 1992.

[10] M. Holland, G.A. Gibson, D.P. Sieworuk: "Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays", *Journal of Parallel and Distributed Databases* 2, 1994.

[11] Jai Menon, Dick Mattson: "Distributed Sparing in Disk Arrays", *Proceedings of the Compcon Conference 1992, San Francisco,* pp. 410-416, 1992.

[12] Arif Merchant and Philipp S. Yu: "Analytic Modeling of Clustered RAID with Mapping Based on Nearly Random Permutation", *IEEE Transactions on Computers,* vol. 45, no. 3, March 1996.

[13] R. Muntz, J. Lui: "Performance Analysis of Disk Arrays under Failure", *Proceedings of the Conference on Very Large Data Bases,* pp.162-173, 1990.

[14] S. Ng, R.L. Mattson: "Uniform Parity Group Distribution in Disk Arrays with Multiple Failures", *IEEE Transactions on Computers,* vol. 44, no. 3, pp. 419-433, March 1995.

[15] D.A. Patterson, G.A. Gibson, R.H. Katz: "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings, ACM SIGMOD Conference,* pp. 109-116, July 1988.

[16] A.L.N. Reddy, P. Bannerjee: "Gracefully Degradable Disk Arrays", *Proc. 21st International Symposium on Fault Tolerant Computing,* pp. 501-506, April 1994.

[17] E.J. Schwabe, I.M. Sutherland: "Improved Parity-Declustering Layouts for Disk Arrays", *Proceedings of the Symposium on Parallel Algorithms and Architectures,* Cape May, N.J., June 1994, pp. 76-84, 1994.

[18] T.J.E. Schwarz, W.A. Burkhard, J. Steinberg: "Permutation Development Data Layout (PDDL) Disk Array Declustering," *UCSD technical report CS98-584,* April 1998.

Appendix

The material, presented here for the reviewers, details and expands upon several points from the body of the paper. The first two are additional experimental results. Most of this material will ultimately appear only within our technical report [18].

1. Response times.
2. Seek counts for fault-free and degraded write operations.
3. PDDL mathematical underpinnings.
4. n is a power of 2 example.
5. Virtual disk interface.
6. n = 55 group of satisfactory base permutations.

PDDL schemes have a strong link with combinatorial mathematics. In a general form, PDDL uses an arbitrary amount of spare space. Then, PDDLs using a solitary base permutation are equivalent to a *difference family.* The set of disks $\mathcal{D} = \{ 0, 1, 2, \ldots, n-1 \}$ is partitioned into g blocks $\{ \mathcal{B}_i \mid 1 \leq i \leq g \}$ each of size k and one additional, possibly empty, block $\mathcal{B}_0$.

$$\begin{aligned} \mathcal{D} \quad &= \mathcal{B}_0 \sqcup \mathcal{B}_1 \sqcup \cdots \sqcup \mathcal{B}_g. \\ \mid \mathcal{B}_i \mid \quad &= k \text{ for all } i = 1, \cdots g \\ \{\mathcal{B}_i \mid 1 \leq i \leq n\} \quad & \text{is a difference family in } \mathcal{D} \,/\, \mathcal{B}_0. \end{aligned}$$

The block $\mathcal{B}_0$ contains the spare space. The base permutation is obtained by enumerating the elements of $\mathcal{B}_0$, then $\mathcal{B}_1$, and so on. The mapping function utilizes addition from $GF\,(\,n\,)$.

Our PDDL scheme is also related to *near resolvable designs* [7, 8]. A PDDL with a solitary base permutation gives rise to a near resolvable design but unfortunately a near resolvable design need not provide a PDDL. There are lengthy lists however of nearly resolvable designs [8].

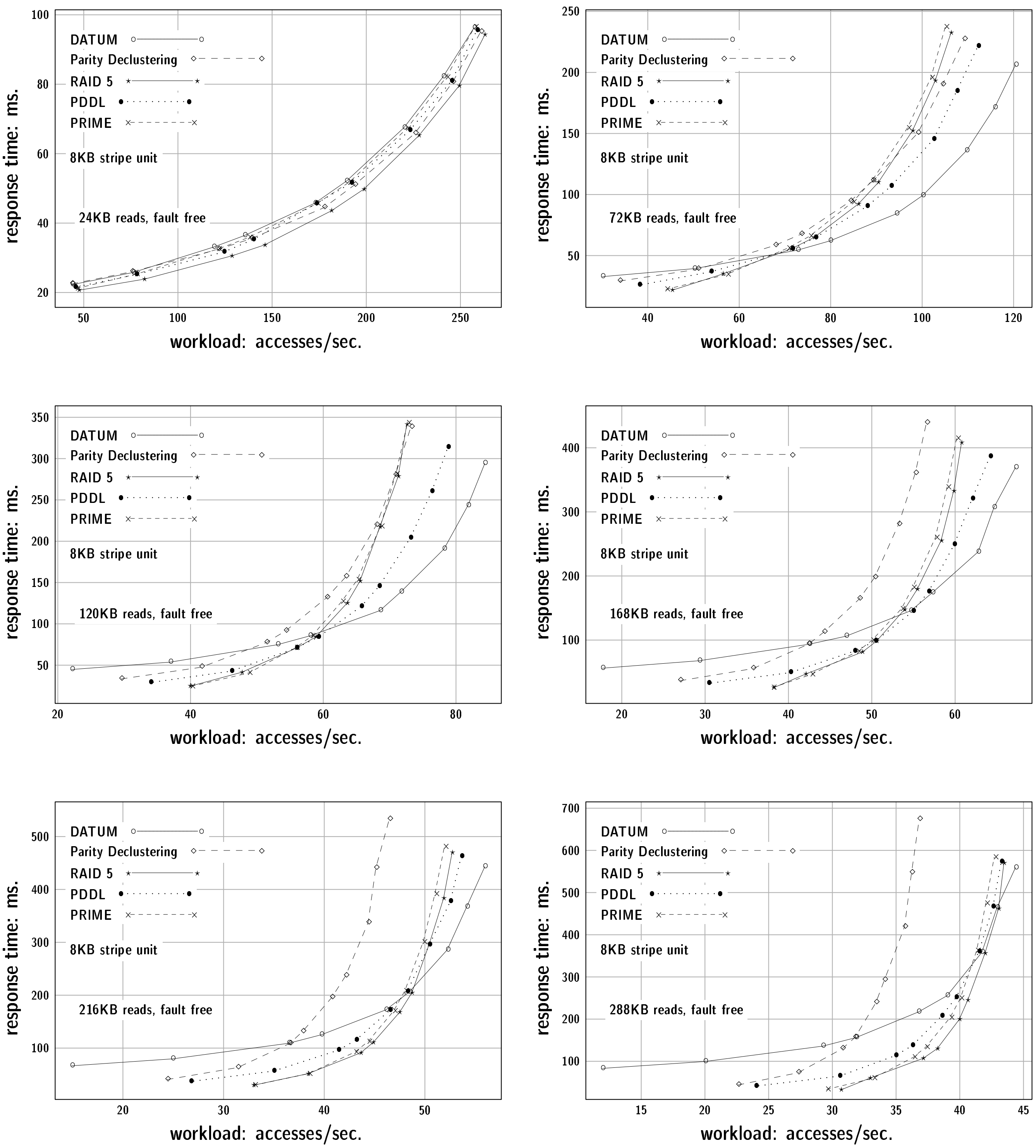

Figure 10: *Read response times: Failure-free mode.*

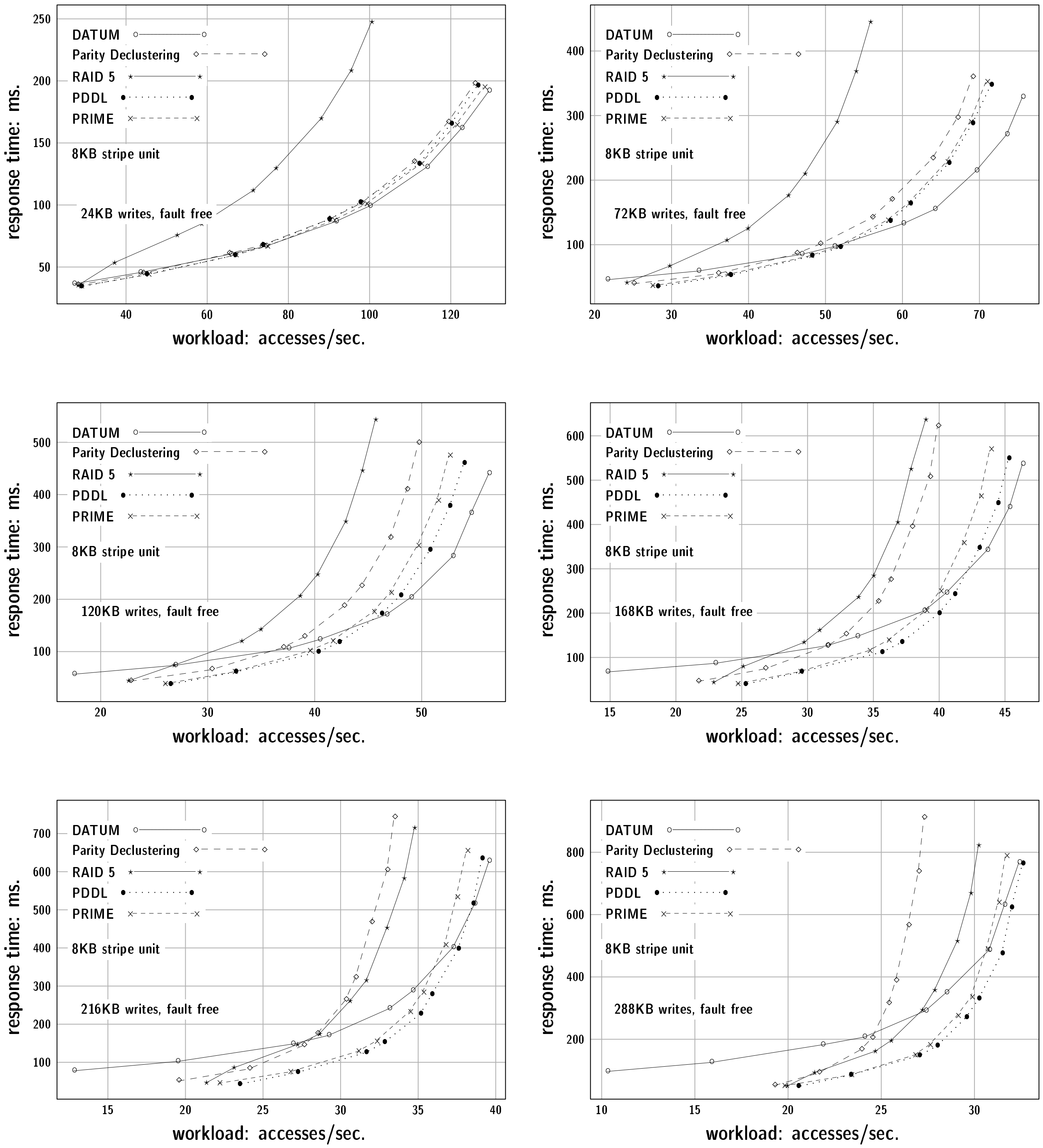

Figure 11: *Write response times: Failure-free mode.*

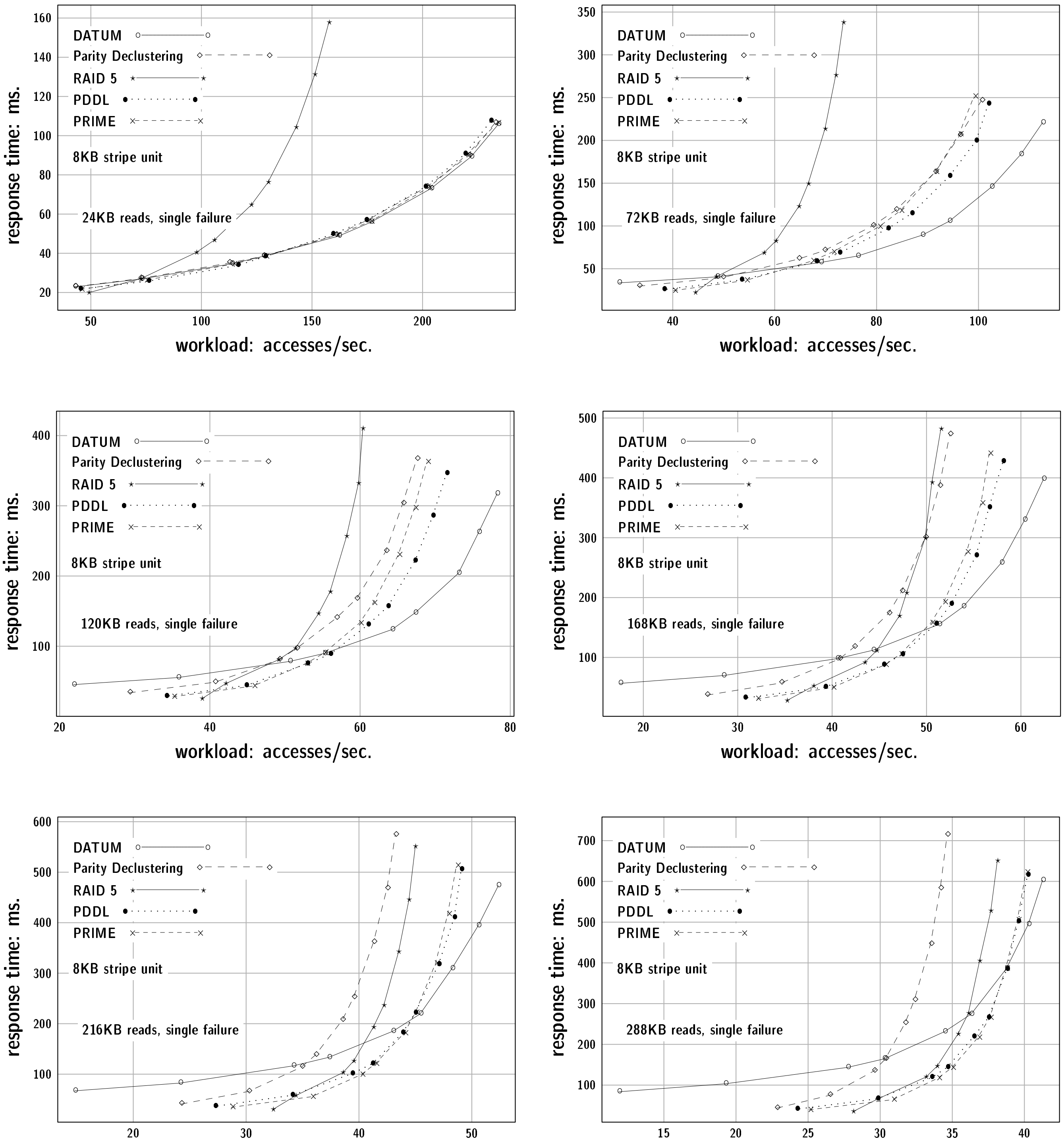

Figure 12: *Read response times: Single failure mode.*

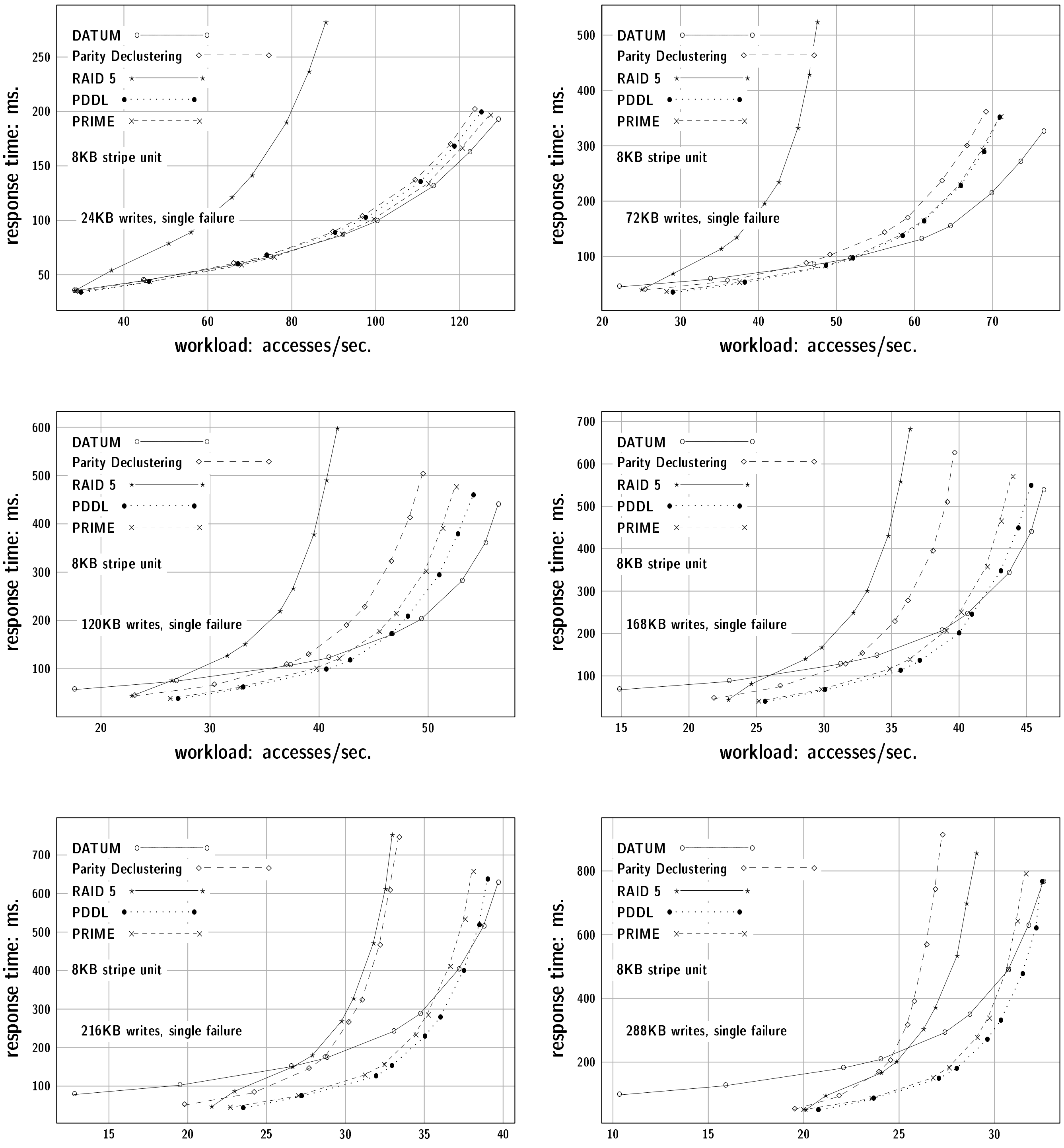

Figure 13: *Write response times: Single failure mode.*

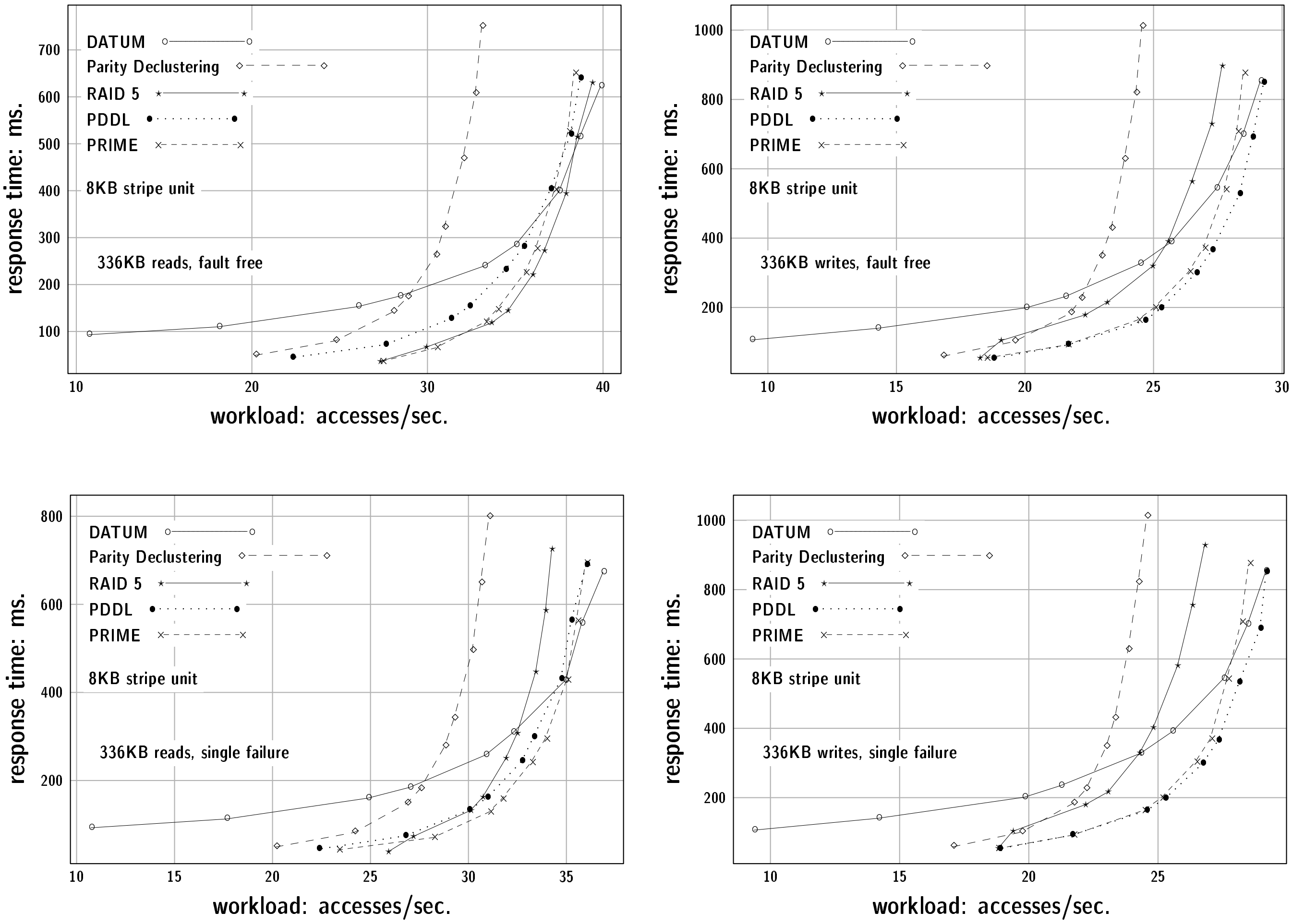

Figure 14: *Response times: 336 KB accesses.*

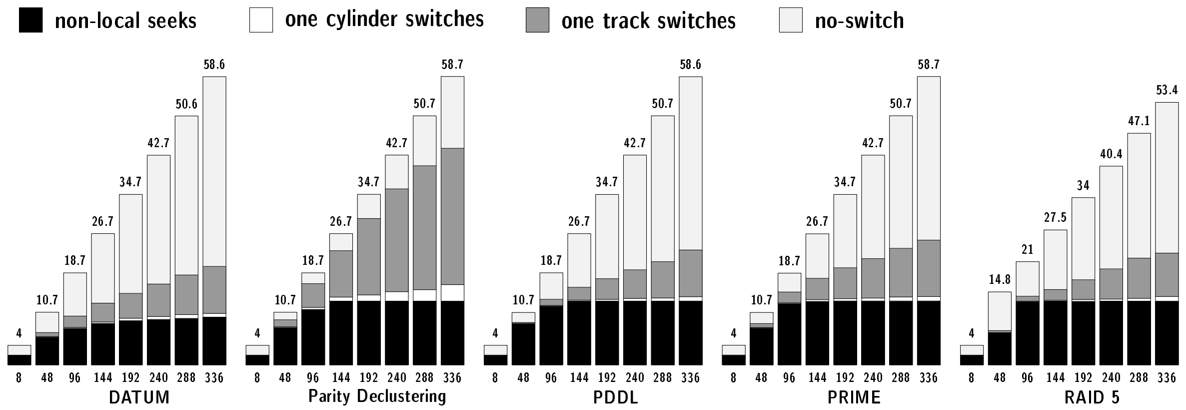

Figure 15: *Fault free write; seek and no-switch counts*

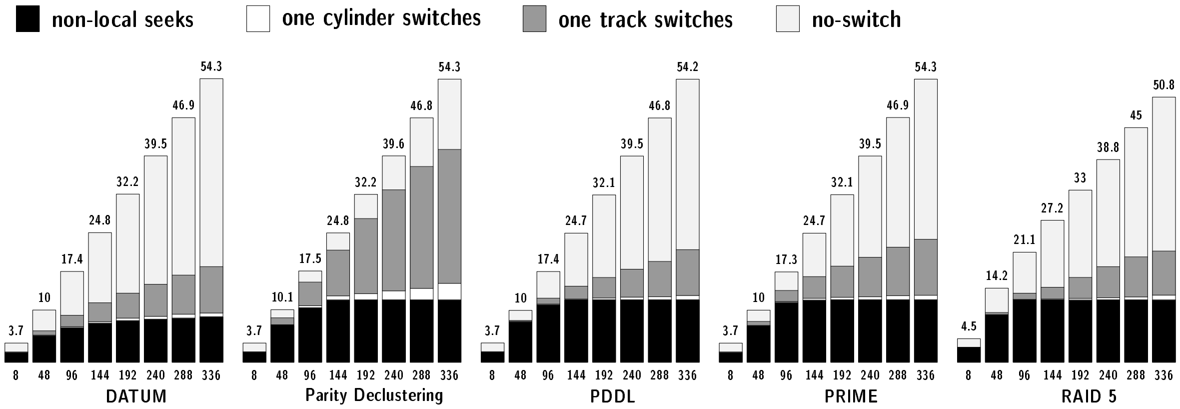

Figure 16: *Degraded write; seek and no-switch counts*

When n is a power of 2, we obtain an extremely efficient mapping implementation, if not the most efficient. We obtain the base permutation via our construction algorithm of section 3. As an example, for $n = 16$ and $g = 3$, we use primitive element $x + 1$ with irreducible polynomial $x^4 + x^3 + x^2 + x + 1$ within $GF(16)$. Successive powers of the primitive element, represented as four-bit integer values, are 1, 3, 5, 15, 14, 13, 8, 7, 9, 4, 12, 11, 2, 6, 10. Thus, the base permutation is as follows.

$$0,\ 1,\ 15,\ 8,\ 4,\ 2,\ 3,\ 14,\ 7,\ 12,\ 6,\ 5,\ 13,\ 9,\ 11,\ 10$$

Our mapping function utilizes exclusive-or as the addition operation with the four-bit values; the mask `0xf` maintains the four-bit width.

```
int permutation[ ] = { 0, 1, 15, 8, 4, 2, 3, 14, 7, 12, 6,
                                      5, 13, 9, 11, 10 } ;

int virtual2physical( int disk , int offset )
{
       return ( ( permutation[disk] ^ offset ) & 0xf ) ;
}
```

We mentioned previously the virtual disk interface. In this interface, the user is aware only of a linear address space for storage. Stripe units have indices 0 through MAXSTRIPEUNIT-1. Our PDDL mapping function is preceeded by the virtualDisk function.

```
int offset , disk ;

int virtualDisk ( int stripeUnit )
{
       offset = stripeUnit / ( g * ( k-1 ) ) ;
       disk = stripeUnit % ( g * ( k-1 ) ) ;
       disk = 1 + disk + disk / ( k-1 ) ;
}
```

0	1	18	24	31	40	48
	2	3	7	11	13	44
	4	19	23	29	32	47
	5	21	30	33	36	53
	6	17	28	49	52	54
	8	12	14	22	34	35
	9	10	20	25	39	46
	15	16	37	42	50	51
	26	27	38	41	43	45

0	1	2	8	25	46	54
	3	6	27	32	41	49
	4	11	26	39	43	45
	5	18	22	24	36	50
	7	10	13	28	40	52
	9	17	20	30	48	53
	12	31	37	38	42	47
	14	16	21	29	44	51
	15	19	23	33	34	35

Figure 17: *Two permutations provide satisfactory base permutations for 55 disks and stripe width six.*

In many configurations, more than a solitary base permutation is evidently required to obtain evenly distributed reconstruction workload. We have already seen such an example for ten disks in section 2. Figure 17 contains a pair of larger permutations for $n = 55$ and $g = 9$.

The read reconstruction and post-reconstruction response times as well as the fault free response times are presented in Figure 18; the sparing has the greatest impact for access sizes close to the stripe size. For all access sizes larger than 72KB, the relative positions of the curves remains similar to those of the 72KB accesses.

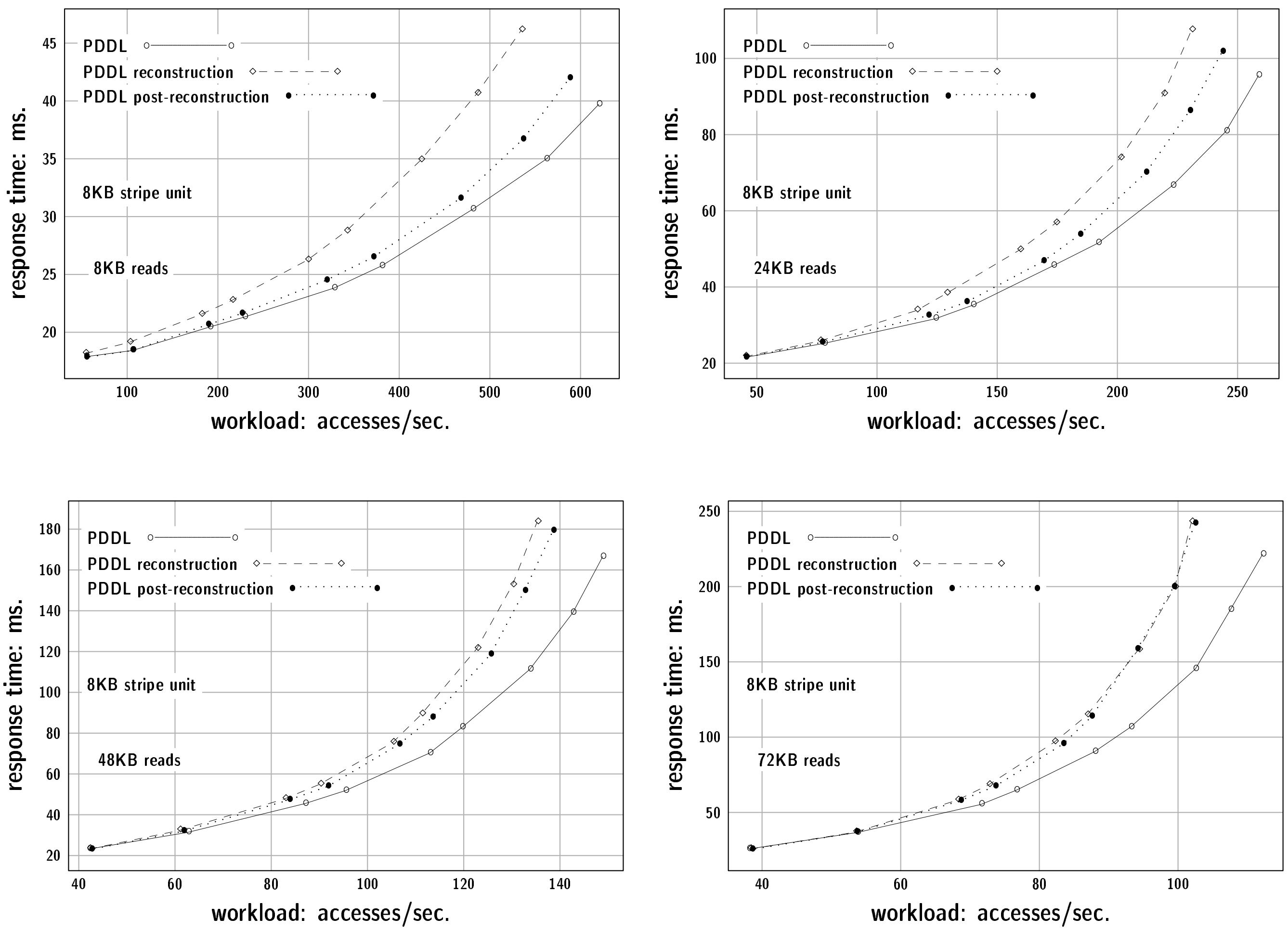

Figure 18: *PDDL Read response times: fault free, reconstruction, and post-reconstruction.*