Chapter 4

A Self-Scaling Benchmark

4.1. Overview

Willy is a flexible, general I/O workload generator. But, by itself, it is not the ideal I/O benchmark. This is because regardless of what set of values are assigned to Willy's parameters, it suffers from the classic I/O benchmarking problems: it does not provide understanding into system performance; it does not scale over current or future systems; it does not allow comparisons of systems, except for the measured workload; and each workload is relevant to only a narrow range of applications. Over time, poor scaling renders benchmarks irrelevant—what was an I/O-intensive program this year may not be I/O-intensive next year.

In this chapter, I describe a method that addresses these shortcomings. The result is an evaluation tool that automatically explores the space of possible workloads, running Willy to measure performance for each workload. In the process of exploring the workload space, the benchmark accomplishes two major goals:

- It shows how performance depends on each parameter and reports which workloads perform well. This increases one's understanding of the system being measured.
- It automatically scales itself by choosing which workloads to report based on each system's performance capabilities. This scaling helps ensure the benchmark's long-term relevance. Unfortunately, scaling also results in running different workloads for different systems, making it difficult to compare two systems directly. Chapter 5 addresses the problem of comparing two systems by using this chapter's results to estimate performance for arbitrary workloads.

This chapter describes how the self-scaling benchmark accomplishes these goals, giving examples and implications along the way. In addition to the workstations benchmarked in Chapter 2, I also measure a Convex C240 mini-supercomputer, a Solbourne 5E/905 file server, a raw-I/O interface on the SPARCstation 1+, two client-server configurations (Sun's Network File System and HP's Distributed Unix), and a beta release of an unannounced workstation.

This chapter is divided into two parts. In the first part, Sections 4.3-4.4, I describe my first attempt to create a self-scaling benchmark, using a ‘‘knee of the curve’’ concept to scale all the parameters. This approach, which I call the *Global-Knee Self-Scaling Benchmark*, is quite useful but has several problems. In the second part, Section 4.5, I revise the method and back off from scaling all parameters using the knee of the curve. Instead, I scale a subset of the parameters using the midpoints of each parameter's range.

4.2. Single Parameter Graphs

Most current benchmarks described in Chapter 2 report the performance for a single workload only. The better benchmarks report performance for multiple workloads, usually in the form of a graph. TPC-B and Sdet, for example, report how performance varies with load (Figure 2.1 on page 23 and Figure 2.2 on page 25). But even these better benchmarks do not show in general how performance depends on parameters such as request size or the mix of reads and writes.

The main output of Chapter 4's self-scaling benchmark is a set of performance graphs, one for each parameter (uniqueBytes, sizeMean, readFrac, processNum, and seqFrac) as displayed in Figure 4.1. While graphing one parameter, all other parameters remain fixed. I call the value at which a parameter is fixed while graphing other parameters the *focal point* for that parameter. The vector of all focal points is called the *focal vector*. In Figure 4.1, for example, the focal vector is {uniqueBytes = 21 MB, sizeMean = 10 KB, readFrac = 0, processNum = 1, seqFrac = 0}. Hence, in Figure 4.1a, uniqueBytes is varied while the other parameters remain fixed (sizeMean = 10 KB, readFrac = 0, processNum = 1, and seqFrac = 0).

Figure 4.2 illustrates the workloads reported by one set of graphs for a three parameter workload space. Although these graphs show much more of the entire workload space than do current benchmarks, they still show only single parameter performance variations; they do not display dependencies between parameters. Unfortunately, a complete exploration of the entire five dimensional workload space requires far too much time. For example, an orthogonal sampling of six points per dimension requires 6^5, or almost 8000, points. On the Sprite DECstation, each workload takes approximately 10 minutes to measure; thus, 8000 points would take almost 2 months to gather! In contrast, measuring six points for each graph of the five parameters requires only 30 points and 5 hours. The usefulness of these single parameter graphs depends entirely on how accurately they characterize the performance of the entire workload space. Chapter 5 will show that the *shapes* of these performance curves are relatively independent of the specific values of the other parameters. The rest of this chapter discusses how to scale the benchmark by choosing the focal vector.

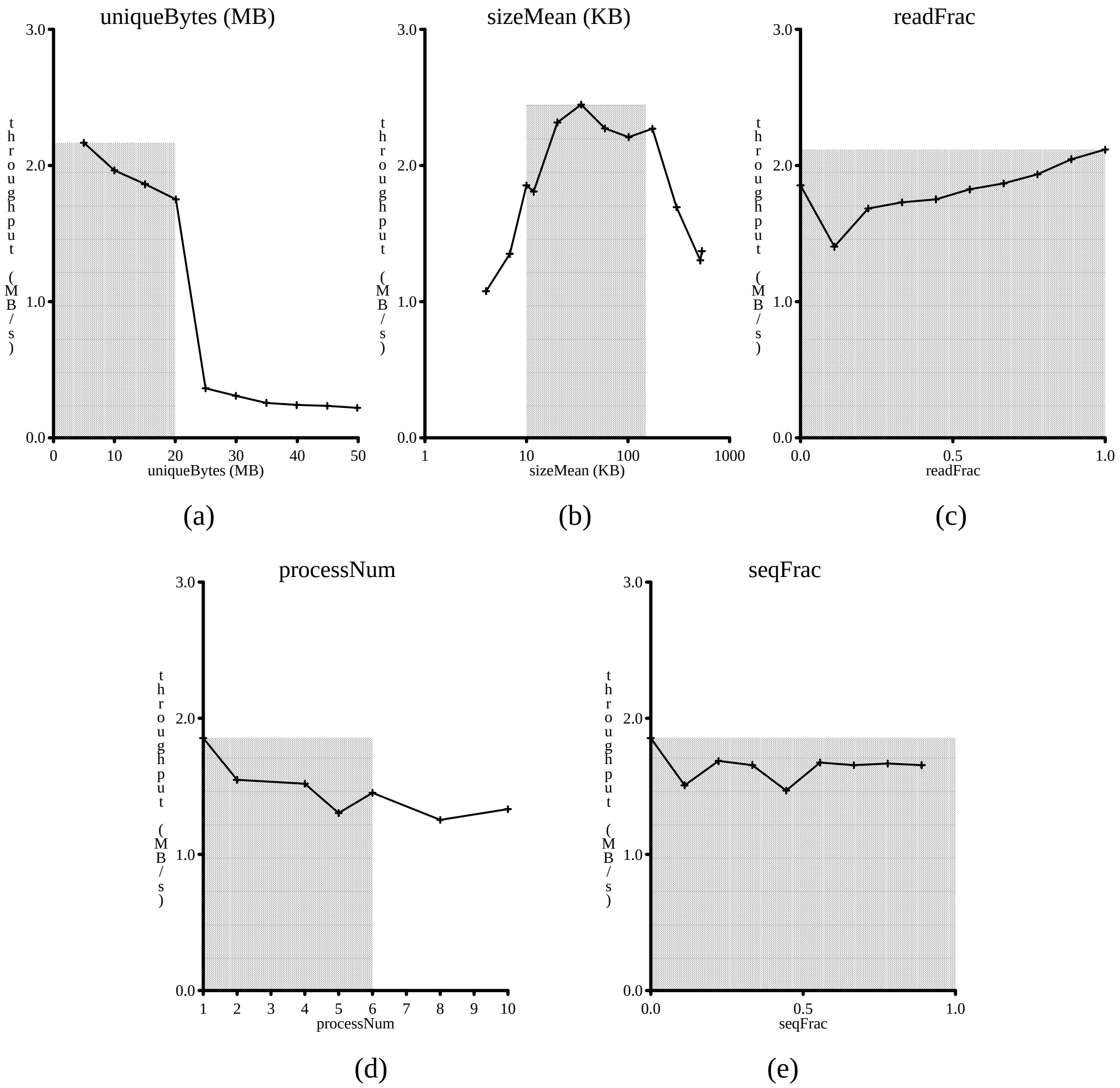

Figure 4.1: Results from a self-scaling benchmark that scales all parameters. This figure shows the results from a self-scaling benchmark of a SPARCstation 1 with 28 MB of memory and a single, SCSI disk. The self-scaling benchmark reports performance for many I/O workloads, where each workload is measured by running Willy with various parameter values. The benchmark reports the range of workloads, shown as the shaded region, that perform well on this system. For example, this SPARCstation performs well if the total number of unique bytes touched is less than 20 MB. It also shows how performance varies with each workload parameter. Each graph varies exactly one parameter, keeping all other parameters fixed at their focal point. For these graphs, the focal point is the same as the global knee point. The knee point for each parameter is defined to be the least restrictive workload value (see Section 4.3) that yields at least 75% of the maximum performance. The range of workloads that perform well (shaded area), is defined as the range of values that yields at least 75% of the maximum performance. The knee points chosen by the benchmark for each parameter are uniqueBytes = 21 MB, sizeMean = 10 KB, readFrac = 0, processNum = 3, seqFrac = 0.

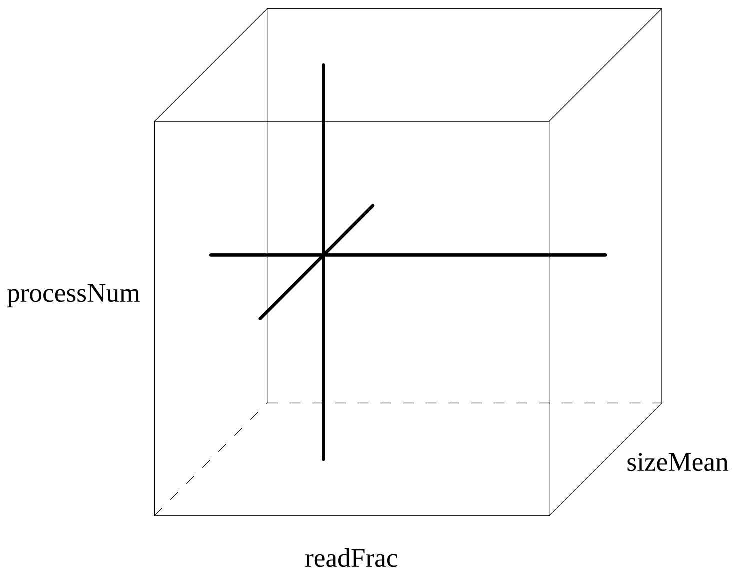

Figure 4.2: Workloads reported by a set of single parameter graphs. This figure illustrates the range of workloads reported by a set of single parameter graphs for a workload of three parameters.

4.3. The Knee Point

A common concept in engineering is that of the *knee of the curve*, or a *knee point*. This phrase usually refers to an intuitive balance point between higher benefit and higher cost. In the realm of I/O performance, benefit refers to higher performance achieved by different values of workload parameters, while cost refers to the difficulty in creating workloads with those values.

Most of Willy's workload parameters have well-defined directions of higher performance and higher cost. Take sizeMean as an example. Larger requests result in higher performance by amortizing any per-request overheads over more bytes. On the other hand, large request sizes are usually difficult to achieve due to practical constraints such as internal fragmentation, long latencies to the last byte, and small file sizes. This tradeoff between practical constraints and higher performance typifies the engineering tradeoffs needed to find a knee point that will provide good performance at reasonable cost. The following is a list of the directions of higher performance and higher cost for Willy's workload parameters:

- uniqueBytes—It is harder to achieve smaller uniqueBytes than larger one. But, smaller uniqueBytes yield higher performance than do larger ones. As a result, programmers often expend much effort to recode their programs in such a way as to work within smaller localities to allow the working set to fit within available memory.
- sizeMean—Large sizes are harder than small sizes and leads to higher performance, as explained above.
- readFrac—ReadFrac is the only parameter that has no clear direction of higher cost or higher performance. Some scientific applications try to minimize reads, since reads force the user to wait for the data. In contrast, applications such as databases try to minimize writes, since writes must be committed to disk. I assume that whichever direction yields higher performance is also the direction of higher cost.
- processNum—Workloads with more processes are harder to achieve than less processes. It is easy to allow only one process at a time to make I/O requests, increasing workload's I/O concurrency is not easy. Increasing the number of processes often increases performance by making use of available parallelism in the disk or memory system. More processes than necessary to achieve maximum throughput is undesirable, however, because response times would degrade without improving throughput.
- seqFrac—Higher seqFrac is harder but leads to higher performance. Many applications naturally generate sequential accesses [Ousterhout85], but, in general, it is easier to generate arbitrary access patterns without regard to sequentiality.

For all parameters, except perhaps readFrac, the direction of higher difficulty is also the direction of higher performance. Murphy's Law strikes again!

It is very difficult to derive a mathematically satisfying definition of the knee of a curve. When I started this research, I expected to find a formula, related to the first derivative of the

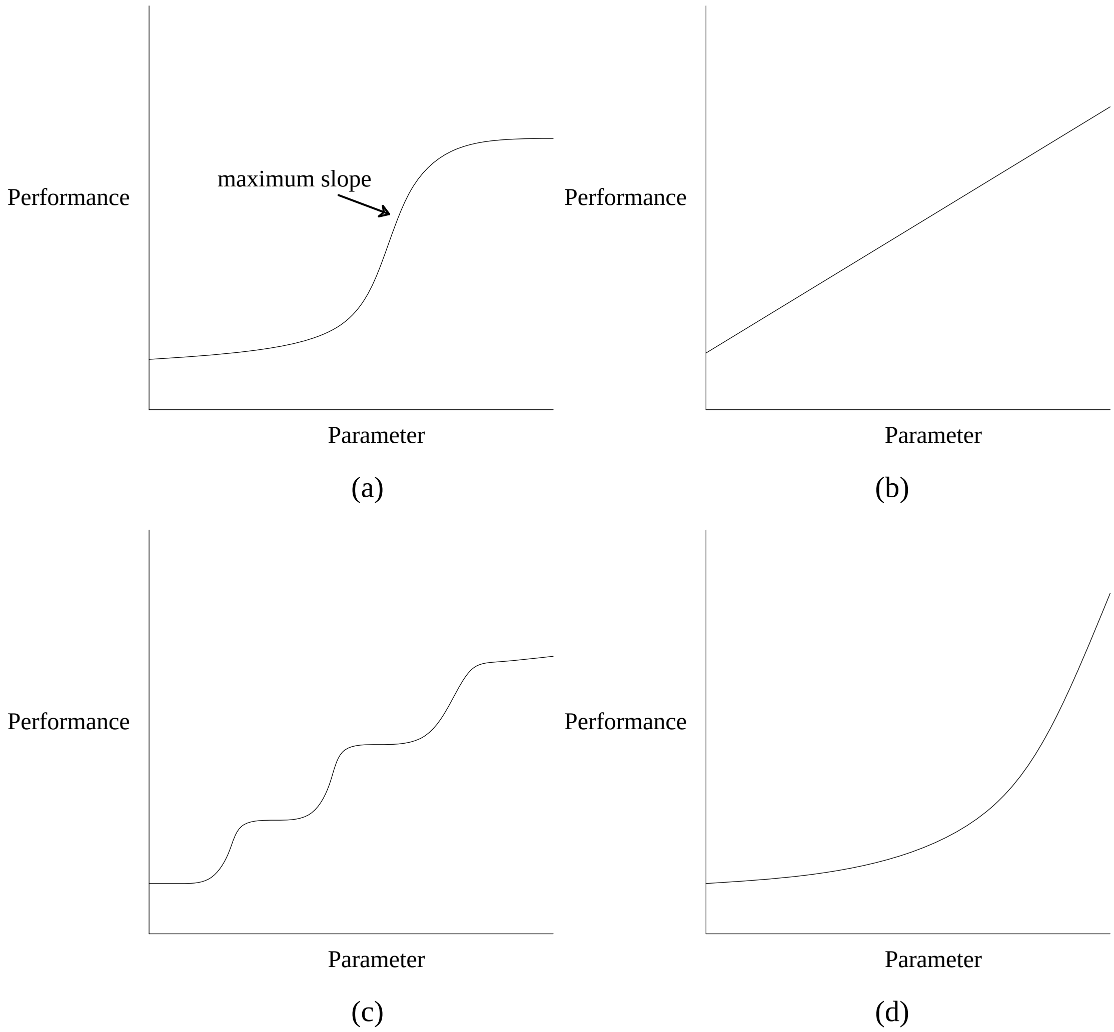

Figure 4.3: Difficulty in defining the knee. Differently shaped performance curves make it difficult to define a universal formula for the knees of curves.

performance, that would define a unique point at which the marginal benefits decreased. Unfortunately, no such formula exists for arbitrarily-shaped performance curves. For the shape of some curves, the choice is fairly easy. Figure 4.3a shows a curve whose knee can be defined as the point of maximum slope. The choice, however, is much less clear for Figure 4.3b, 4.3c, and 4.3d. One stumbling block to defining the knee in terms of the slope of the performance curve is the scale of the axes. This is because the scale of most parameter's axes is defined independently from the scale of the performance axis. Choosing a knee according to the value of the slope would hence lead to different knees for different axes scales. In the absence of any aesthetically pleasing mathematical definition, I chose an engineering solution: I define the knee of the curve as the point that reaches 75% of the maximum performance. Most of the shortcomings of this knee approach to scaling, listed in Section 4.4.2, will hold regardless of the actual fraction used.

The knee point of Willy's workload parameters provides a hint of how a system might be used. This positive correlation between a system's knee and the use of a system may occur because people who care about performance will buy a computer system that performs well for their workload or modify their current one to perform better for their workload. A few people may even re-code their application to better utilize their system's capabilities. The following are some examples.

- If a potential buyer of a disk system anticipates many concurrent I/Os, he is more likely to buy and use a disk array, which supports higher concurrency (shown by a larger knee value in the processNum performance graph), than a single-disk system.
- If an important program, or class of programs, is thrashing on the system due to a small file cache, the user of that system will likely buy more memory to allow the program to fit in the cache. The RAID-II project [Lee92], for example, bought more memory for the SPARCstation that routed the board to prevent thrashing.
- Programmers for a small class of applications will be able to recode their programs to take advantage of potential performance improvements in moving toward the knee point. Designers of operating systems can increase the block size of the file system; databases can perform parallel queries to increase concurrency [Hong91]; scientific application writers can recode their matrix reference patterns to stay within the file system cache [Kim87].

For some parameters, users will probably always run workloads with a variety of values on any given system. For instance, they will likely always run applications that both read and write data in varying proportions.

The knee point concept can be extended to multiple parameters. I define *global knee point* to be a point at which all parameters are simultaneously at their knee point. As shown below, at least one global knee must exist, assuming the knee points for each parameter are continuous over the workload space.

For a workload with two parameters, X and Y, I can find all possible knee points for X with Y fixed at y. If I do this for all values of Y between y_{min} and y_{max}, I graph a curve, such as the solid line shown in Figure 4.4. Note that because knee points are continuous, this curve is continuous and exists for all points between y_{min} and y_{max}. The same procedure can be carried

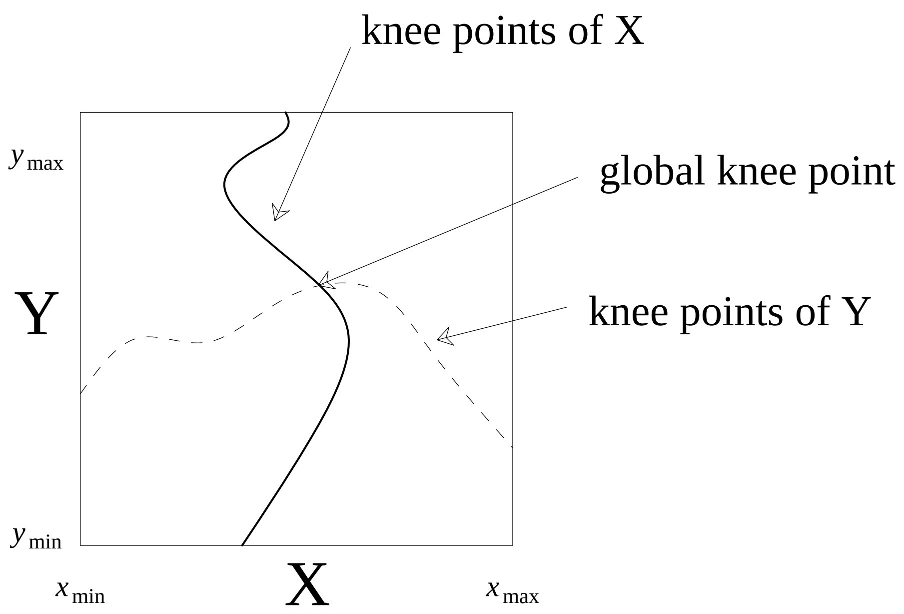

Figure 4.4: Global knee point with two parameters. This figure proves the existence of a global knee point with two parameters, X and Y. Assuming knee points are continuous, the knee point curve for each parameter must cross in at least one place.

out to find all knee points for Y for X fixed at all values between x_{min} and x_{max}.

For a workload with three parameters, X, Y, and Z, the same can be shown. I can find the knee point of Z for each combination of (X,Y), which appears graphically as a smooth surface over the entire X-Y plane. Because the surface covers the entire X-Y plane, it must intersect with a similar surface for Y's knee point. Intersecting all three knee point surfaces results in at least one global knee point. The next section describes how to find and use this global knee point to create a self-scaling benchmark.

4.4. A Global Knee Self-Scaling Benchmark

A self-scaling benchmark is one that adjusts the workloads that it runs and compiles a report based on the capabilities of the system being measured. Sdet and TPC-B both do this for one aspect of a workload, that is, load (processNum) [SPEC91, TPCB90]. Sdet reports the maximum throughput, which occurs at different loads for different systems. TPC-B reports maximum throughput subject to a response time constraint; this also occurs at different loads for different systems. In this section, I describe a benchmark that generalizes this scaling concept to all workload parameters.

One way to create a self-scaling benchmark is to set the focal vector to be the same as the global knee point. I call this version of the self-scaling benchmark the *global-knee, self-scaling benchmark.*

The global-knee, self-scaling benchmark uses a simple, iterative approach to finding the global knee point. As it runs, it fixes all but one parameter and finds the knee for that one parameter. It then fixes the value for that parameter at the newly found knee point and continues on to another parameter. This iterative process continues until all parameters are simultaneously at their knee points. Although this process could theoretically never converge, in reality very few iterations are needed (only four were needed in the example below). This quick convergence is due to the shape, and thus the knee, of one parameter's curve being reasonably independent of the other parameters. This independence is used and verified in Chapter 5.

After finding the global knee point, the global-knee, self-scaling benchmark reports two items. First, it reports which ranges of workoads perform well on the system. Second, it returns a graph of performance versus each parameter with the other parameters fixed at their focal points. This set of graphs helps the user to understand the performance of the system by showing how performance varies with each parameter.

4.4.1. Example Run of Global-Knee, Self-Scaling Benchmark

This section presents results from a run of the global-knee, self-scaling benchmark. The system being measured is the one-disk SPARCstation of Table 2.1. In Figure 4.1 on page 52, the shaded region on each graph is the range of workloads that perform well for this system, that is, the workload values that yield at least 75% of the maximum performance. When a parameter is varied, the other parameters are fixed at their global knee points chosen by the benchmark {uniqueBytes = 21 MB, sizeMean = 10 KB, readFrac = 0, processNum = 1, seqFrac = 0}. These are the least restrictive values in the range of workloads that perform well. Some of the conclusions that the global-knee, self-scaling benchmark helps us reach about this system are as follows:

- The effective size of the file cache is 21 MB. Applications that have a locality space larger than 21 MB will go to disk frequently.
- I/O workloads with larger average sizes yield higher throughput than smaller average sizes. To get reasonable throughput, request sizes should be between 10 KB and 200 KB. This information may help programmers of operating systems in choosing the best block size.
- Workloads consisting mostly of reads perform slightly better than workloads with more writes.

- Increasing concurrency does not improve performance. As expected, without parallelism in the disk system, workload parallelism is of little value.
- Sequentiality does not affect performance at the global knee.

4.4.2. Problems

The global-knee, self-scaling benchmark yields interesting results and helps one understand the system. There are, however, several problems with self-scaling all workload parameters by setting the focal vector to be the same as the global knee. First, the iterative process of finding the global knee may be slow because the number of iterations depends on how truly independent the knee point of one parameter is from the values of the other parameters. Because each iteration takes several hours, more than a few iterations may render the benchmark unusable.

Second, there are really two criteria to consider when choosing the focal vector point for a parameter. Take readFrac as an example. The first criterion is what has been mentioned above—scaling readFrac's value to the value for workloads most likely to run on this system. But there is also a second criterion: *the performance curves while readFrac is at its focal point should apply to workloads where readFrac differs from its focal point*. In other words, the shape of the performance curves at the focal point of readFrac should be representative of the shape of the performance curves at other values of readFrac. When they are, accurate performance prediction can be done based on these curves. This criterion often precludes choosing an extreme value. For example, if reads were 100 times faster than writes, the global-knee, self-scaling benchmark might pick readFrac = 1.0 as the focal point. But readFrac's focal point of 1.0 may yield performance graphs for the other parameters that do not apply to workloads with some writes. For readFrac, the knee of the performance curve may *not* be a good predictor of what workloads are run on a system; hence the criterion of predicting which workloads will be run is not as important as picking a value representative of workloads in general. It would be

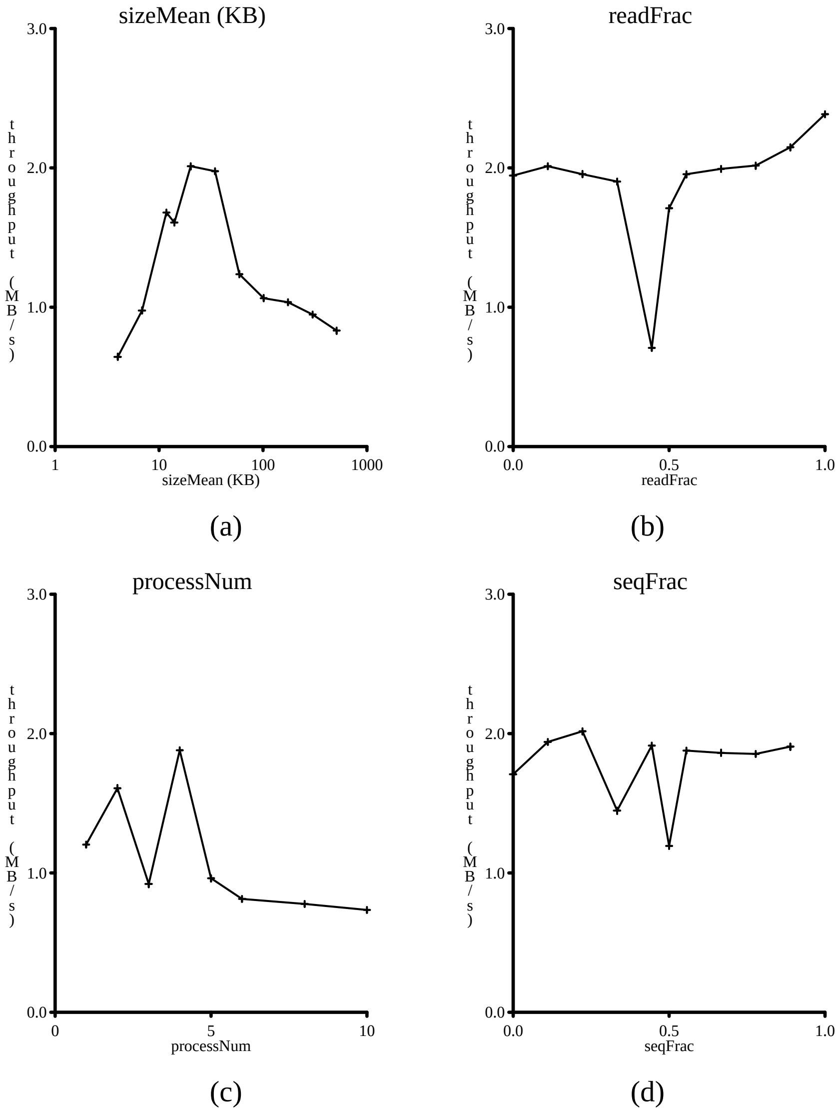

Figure 4.5: Instability in graphs on the border of performance regions. This figure demonstrates the instability of choosing the focal point for uniqueBytes close to the border of two performance regions. The focal point chosen for this family of curves is uniqueBytes = 23 MB, sizeMean = 14 KB, readFrac = .5, processNum = 2, seqFrac = .5.

better to choose an intermediary value rather than the knee as the focal point. In fact, as will be seen in Chapter 5, general applicability is a more important criterion than scaling to the knee point, because if the shape of the performance curve is generally applicable, performance can be estimated for any workload.

The third problem is that the uniqueBytes parameter often has distinct performance regions, which correspond to the various storage hierarchies in the system. In Figure 4.1a, uniqueBytes smaller than 21 MB primarily use the file cache, while uniqueBytes larger than 21 MB primarily use the disk. When the value of uniqueBytes is on the border between the file cache and disk region, performance is often unstable (see Figure 4.5, where uniqueBytes = 23 MB). The knee point is usually on the border between the storage hierarchy regions, so choosing the focal point for uniqueBytes to be the knee point makes it likely that performance will be less stable. Graphing in the middle of a hierarchy level's performance region should be more stable than graphing on the border between performance regions. Another problem that arises from the storage hierarchy regions is that each level of the storage hierarchy may give different performance shapes. Figures 4.8 and 4.9 show how, depending on the storage hierarchy region, reads may be faster than writes (Figure 4.8g), slower than writes (Figure 4.9b), or the same speed as writes (Figure 4.8b).

4.5. A Better Self-Scaling Benchmark

There are a variety of solutions to the problems listed above. For the distinct performance regions discovered by uniqueBytes, I measure and report multiple families of graphs, one family for each performance region (Figure 4.1 shows a single family of graphs). These regions are distinguished using a heuristic based on the slope of the uniqueBytes curve. For the SPARCstation 1+, the slope of the uniqueBytes curve is shown in Figure 4.6. The heuristic delineates performance regions based on what values of uniqueBytes are lower than the average slope of the uniqueBytes curve. In Figure 4.6, the area near 20 MB is below the average slope, so the heuristic defines two performance regions: one from 0-20 MB, one from 20-60 MB. The self-scaling benchmark measures and reports one family of graphs for each of these performance regions. For instance, the first family of graphs is shown in Figure 4.7a-d and has uniqueBytes

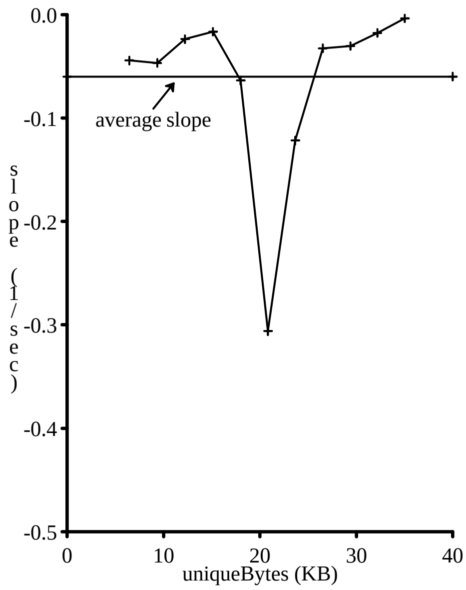

Figure 4.6: Slope of uniqueBytes curve for SPARCstation 1+. This figure plots the slope of the uniqueBytes curve (Figure 4.7). The one area of the graph with slope below the average slope (0.06 per second) is near 20 MB. The self-scaling benchmark uses this information in a heuristic algorithm to delineate performance regions.

= 12 MB. The second family of graphs, Figure 4.7f-i, has a separate focal point with uniqueBytes = 42 MB.

To improve the general applicability of the graphs, I choose the focal point of each parameter to be in the ‘‘middle’’ of the parameter range. With increasing distance from the focal point. the performance shapes’ applicability will degrade. Using the middle of a parameter’s range minimizes the average overall distance from the focal point and should hence maximize the applicability of the focal point. For parameters such as readFrac and seqFrac, the range is easily defined (0 to 1) hence a midpoint of 0.5.[1] The remaining parameters are uniqueBytes, sizeMean, and processNum; for each performance region, the focal point for uniqueBytes is set at the middle of that region, and for sizeMean and processNum, the focal point is set at the

[1]One could also imagine allowing the focal point to stray from the exact middle of the parameter range toward the knee point for that parameter. This might provide a balance between the two criteria for choosing a focal point. I leave this as possible future research.

value that yields a performance halfway between the minimum and maximum.

After these solutions and modifications are made to the global-knee, self-scaling benchmark, the revised program is called simply the *self-scaling benchmark*.

4.5.1. Examples

This section contains results obtained from running the self-scaling benchmark on five systems—the three systems of Table 2.1, plus a four-processor Convex C240 minisupercomputer and a Solbourne 5E/905 file server, described in Table 4.1.

4.5.1.1. SPARCstation 1+

Figure 4.7 shows the results from the self-scaling benchmark on the SPARCstation 1+. The uniqueBytes values that characterized the two performance regions are 12 MB and 42 MB. Graphs a-d show the file cache performance region, measured with uniqueBytes = 12 MB. Graphs f-i show the disk performance region, measured with uniqueBytes = 42 MB. In addition to what was learned from the first self-scaling benchmark, the following can be seen:

- Larger request sizes yield higher performance. This effect is more pronounced in the disk region.

System Name	Convex C240	Solbourne 5E/905
Year Released	1988	1991
CPU	C2 (4 processors)	SPARC (5 processors)
Speed	220 MIPS	22.8 SPECint
Disk System	4 DKD-502 RAID 5	2 Seagate IPI
I/O Bus	IPI-2	IPI-2
Memory Bus	200 MB/s	128 MB/s
Memory Size	1024 MB	384 MB
Operating System	ConvexOS 10.1 (BSD derived)	SunOS 4.1A.2 (revised)

Table 4.1: Description of other machines used in self-scaling benchmark examples. The Convex C240 is a four processor mini-supercomputer. The benchmarked disk system is a four disk RAID 5 with a striping unit of 64 KB The Solbourne is a network file server for workstations using 2 IPI disks and a stripe unit of 64 KB.

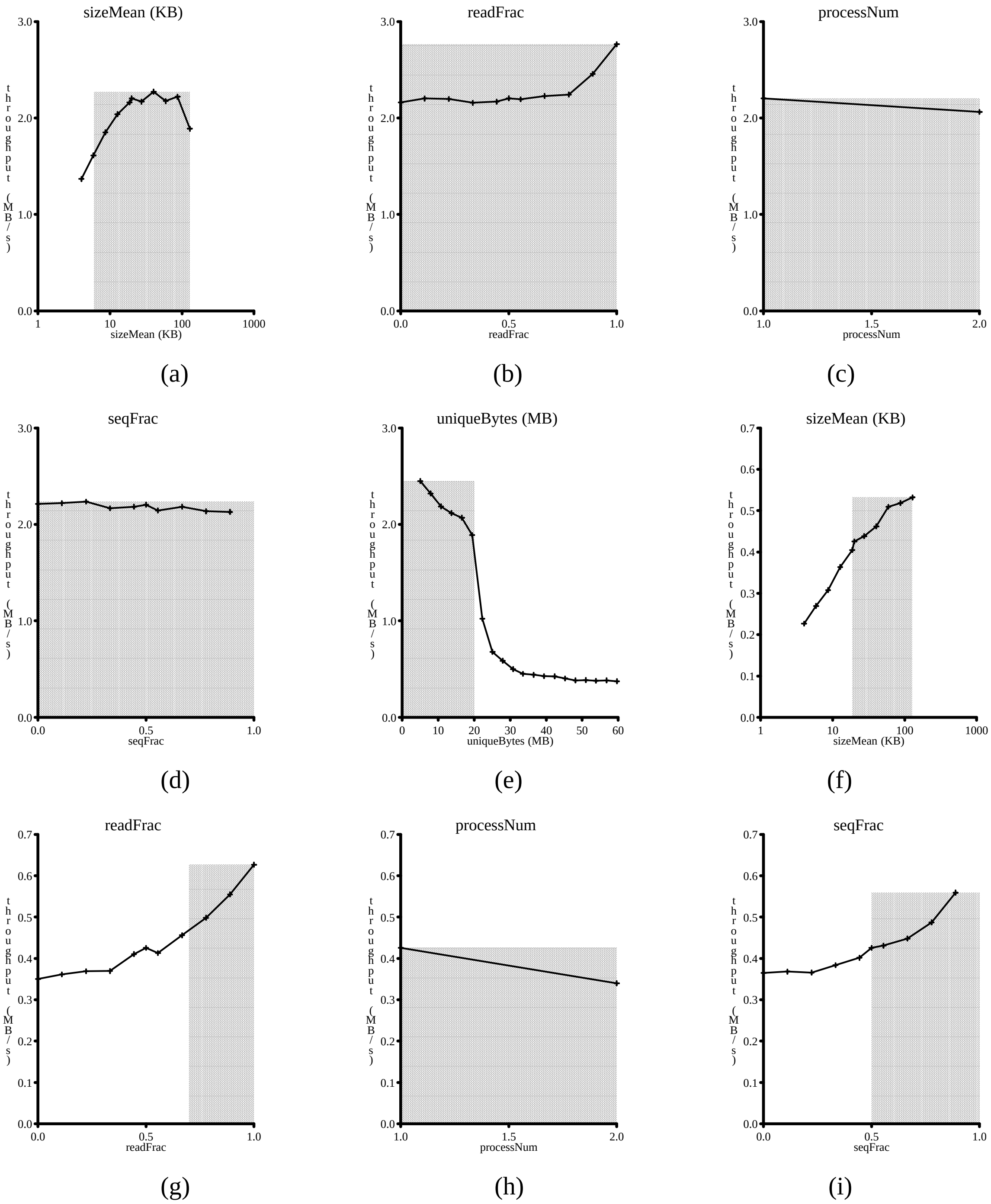

Figure 4.7: Results from a better self-scaling benchmark for a SPARCstation 1+. This figure shows the results from the revised, self-scaling benchmark of the SPARCstation 1+. The focal point for uniqueBytes is 12 MB in graphs a-d and 42 MB in graphs f-i. For all graphs, the focal points for the other parameters is sizeMean = 20 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Increasing sizes improves performance, more so for disk accesses than file cache accesses (Figures a and f). Reads are faster than writes, even when data is in the file cache, because i-nodes must still go to disk. Sequentiality increases performance only for the disk region (Figure i).

- Reads are faster than writes, even when all the data fits in the file cache (Figure 4.1.b). Although the data fits in the file cache, writes still cause i-node changes to be written to disk periodically (for reliability) in case of a system crash.
- Sequentiality offers no benefit in the file cache region (Figure 4.1.b) but offers a substantial benefit in the disk region.

4.5.1.2. DECstation 5000/200

Figures 4.8 and 4.9 show self-scaling benchmark results for the DECstation 5000/200. The uniqueBytes graph (Figure 4.8e) shows three performance plateaus, uniqueBytes = 0 to 5 MB, uniqueBytes = 5 to 20 MB, and uniqueBytes > 20 MB. Thus, the self-scaling benchmark gathers three sets of measurements, one set with uniqueBytes equal to 2 MB (Figure 4.8a-d), one set with uniqueBytes equal to 15 MB (Figure 4.8e-i, and one set with uniqueBytes equal to 36 MB (Figure 4.9). One interesting result is a phenomenon involving readFrac.

In the first performance level (uniqueBytes = 2 MB), reads and writes are the same speed (Figure 4.8b). At the next performance level (uniqueBytes = 15 MB), reads are much faster than writes (Figure 4.8g). This is due to the effective write cache of Sprite's LFS being much smaller than the reads cache, so reads are cached in this range of uniqueBytes while writes are not. The effective file cache size for writes is only 5-8 MB, while for reads it is 20 MB [Rosenblum92].[2] In contrast, when uniqueBytes is large enough to exercise the disk for both reads and writes, writes are faster than reads (Figure 4.9b). This phenomenon is due to Sprite's LFS, which improves write performance by grouping multiple, small writes into fewer, large ones.

---

[2]LFS limits the number of dirty cache blocks to avoid deadlock during cleaning. The default limit was tuned for a machine with 128 MB of memory; in production use, this limit would be changed to match the 32 MB system being tested.

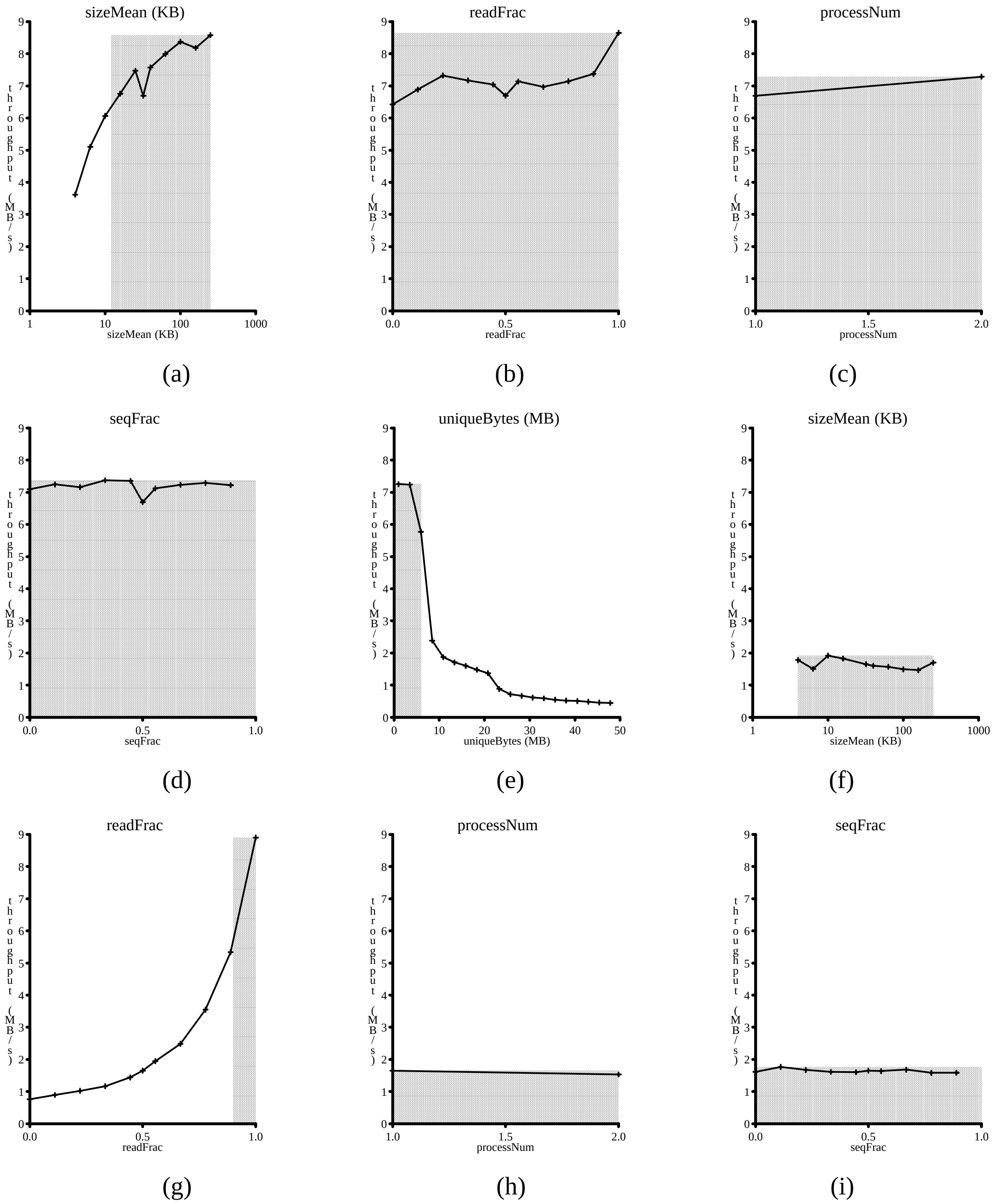

Figure 4.8: Self-scaling benchmark for DECstation 5000/200—Part 1. This figure shows results from the revised, self-scaling benchmark of the DECstation 5000/200. Graph (e) shows three plateaus in uniqueBytes, due to the different effective file cache sizes for reads and writes. Graphs for the third plateau are shown in Figure 4.9. The focal point for uniqueBytes is 2 MB in graphs a-d and 15 MB in graphs f-i. For all graphs, the focal points for the other parameters is sizeMean = 40 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Note how reads can be faster than writes (g), slower than writes (Figure 4.9.b) or the same speed (b).

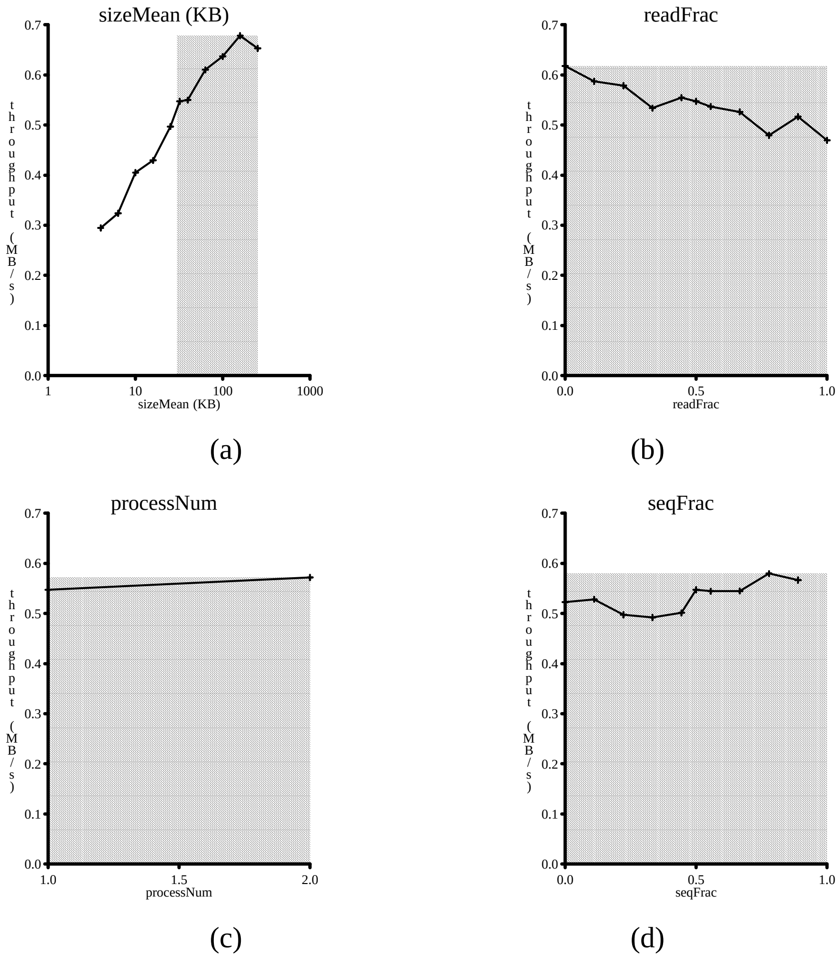

Figure 4.9: Self-scaling benchmark for DECstation 5000/200—Part 2. This figure continues the results from the revised, self-scaling benchmark of the DECstation 5000/200. The focal point for these graphs is uniqueBytes = 36 MB, sizeMean = 40 KB, readFrac = .5, processNum = 1, seqFrac = .5.

4.5.1.3. HP 730

Figure 4.10 shows the results of the self-scaling benchmark for the HP 730. Note how small the effective file cache is. This is due to the HP/UX operating system's choice not to make all memory available to the file cache. In contrast, SunOS maps files into virtual memory, which allows the file cache to fill the entire physical memory. HP/UX, on the other hand, reserves a fixed amount of space, usually 10%, as the file cache. Since this system has 32 MB of main memory, the file cache is approximately 3 MB. The self-scaling benchmark thus uses

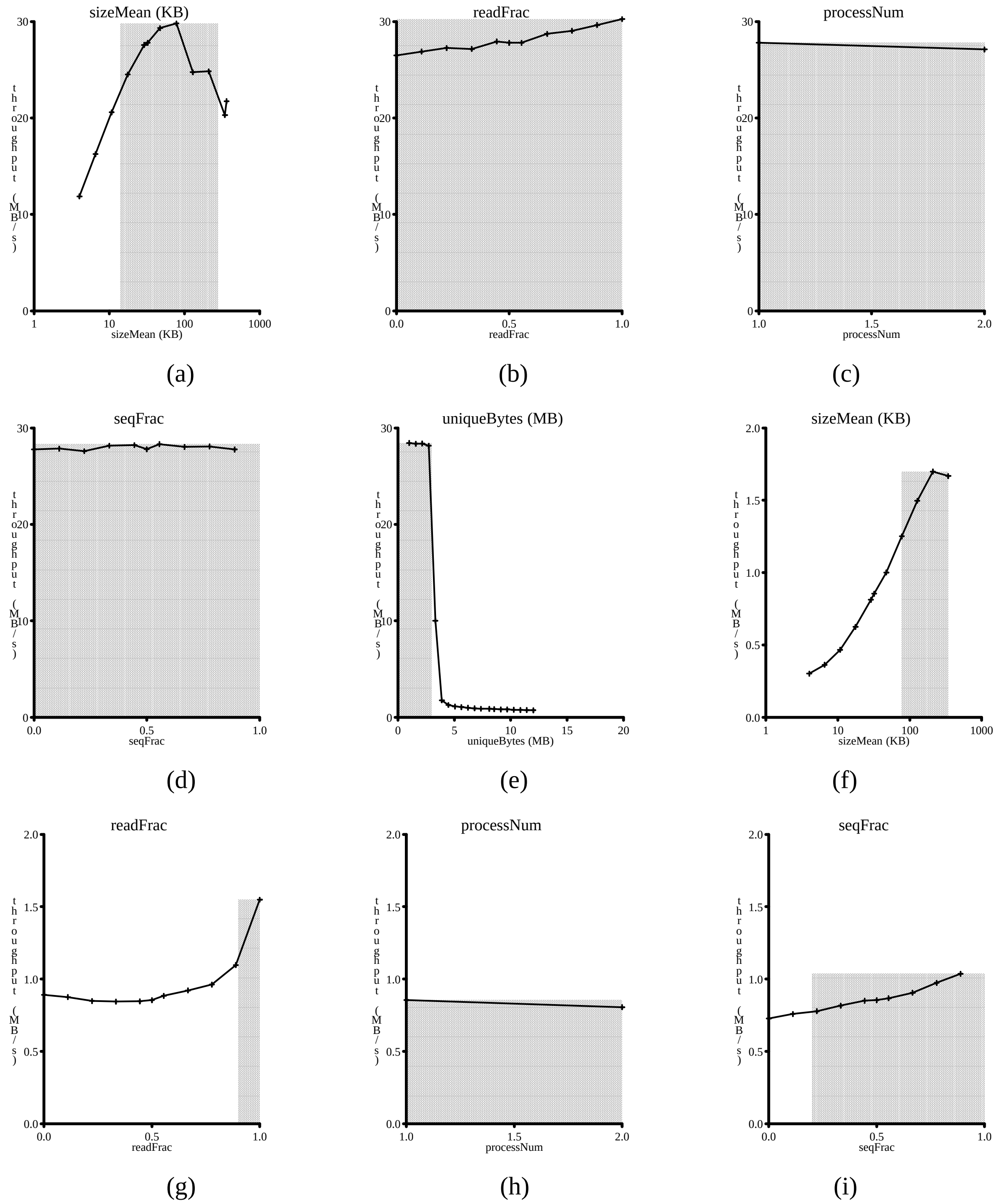

Figure 4.10: Self-scaling benchmark for HP 730. This figure shows the results from the revised, self-scaling benchmark of the HP 730. The focal point for uniqueBytes is 1.8 MB in graphs a-d and 8 MB in graphs f-i. For all graphs, the focal points for the other parameters are sizeMean = 32 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Graph (e) reveals that the file cache is much smaller than the size of main memory would usually indicate (3 MB instead of 32 MB). Also note the extremely high performance, up to 30 MB/s, when uniqueBytes is small enough to fit in the file cache.

two focal points, uniqueBytes = 2 MB and uniqueBytes = 6 MB. Also note the high throughput of the HP 730 when accessing the file cache, peaking at almost 30 MB/s for large accesses (Figure 4.10a). This high performance is due to its fast, interleaved memory system (peak memory bandwidth is 264 MB/s) and to the use of a VLSI memory controller that accelerates cache-memory write backs [Horning91].

4.5.1.4. Convex

Results from the self-scaling benchmark of a Convex C240 are shown in Figure 4.11. The curves are similar to those measured on the SPARCstation 1+, but with three main differences:

- Absolute performance is very high. File cache performance reaches 25 MB/s (Figure 4.11b), while disk performance reaches almost 10 MB/s (Figure 4.11f). This high performance is due to Convex's 200 MB/s memory system and performance-focused (as opposed to cost-performance) implementation. Because the Convex disk system is very fast, the performance difference between the file cache and disk region is smaller than for the other systems.
- The effective file cache for the Convex is 800 MB. This is due to the 1 GB of main memory resident on the computer and an operating system that gives the file cache use of the entire main memory.
- Disk and file cache performance continues to improve with increasing size until requests are 2 MB (Figure 4.11f), while most other computers reach their peak performance with sizes of a few hundred kilobytes.

4.5.1.5. Solbourne

Figure 4.12 shows the self-scaling, benchmark results for the Solbourne 5E/905. Two differences from the other graphs are evident.

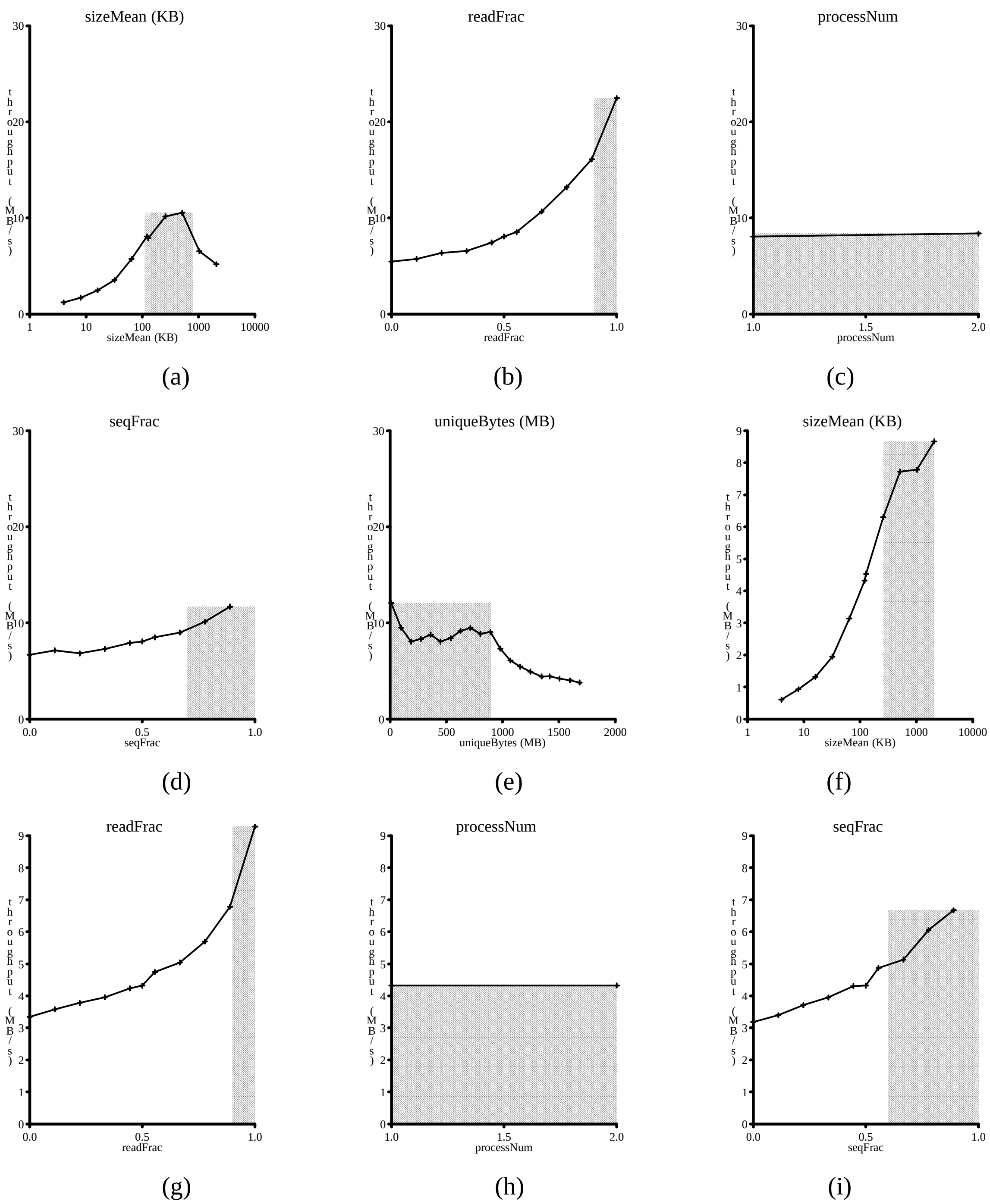

Figure 4.11: Self-scaling benchmark for Convex C240. This figure shows the results from the revised, self-scaling benchmark of the Convex C240. The focal point for uniqueBytes is 450 MB in graphs a-d and 1376 MB in graphs f-i. For all graphs, the focal points for the other parameters are sizeMean = 120 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Note how large the file cache is (800 MB), reflecting how much main memory the system has (1 GB). Also note how large sizeMean grows before reaching its maximum performance (graphs a and f). Because the Convex disk system is very fast, the performance difference between the file cache and disk region is smaller than for the other systems.

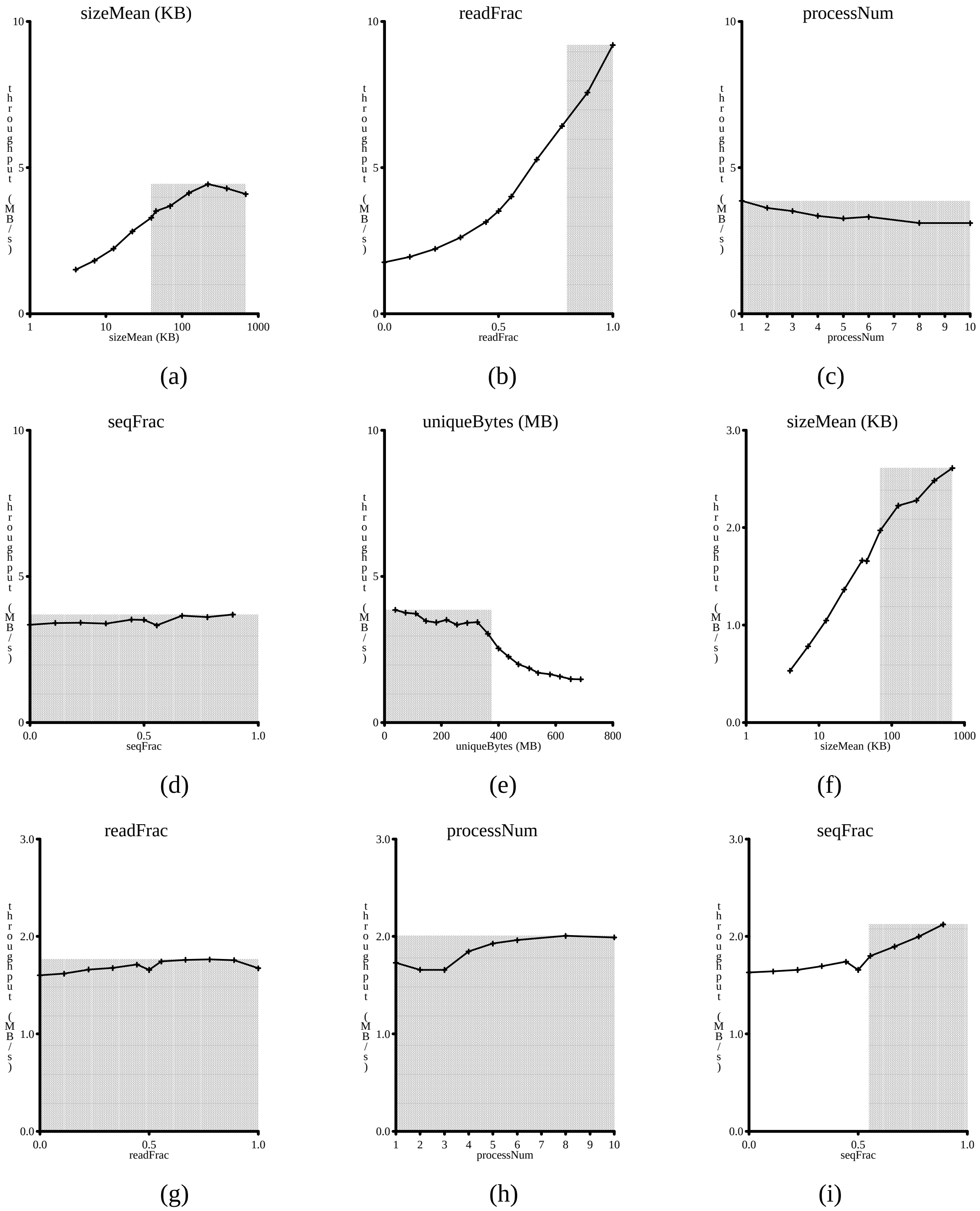

Figure 4.12: Self-scaling benchmark for Solbourne. This figure shows the results from the revised, self-scaling benchmark of the Solbourne. The focal point for uniqueBytes is 180 MB in graphs a-d and 540 MB in graphs f-i. For all graphs, the focal points for the other parameters are sizeMean = 45 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Note how much slower writes are than reads when the file cache is used (graph b). Apparently, the cache write-back policy on the Solbourne causes writes to perform at disk speeds even when all data fits in the file cache.

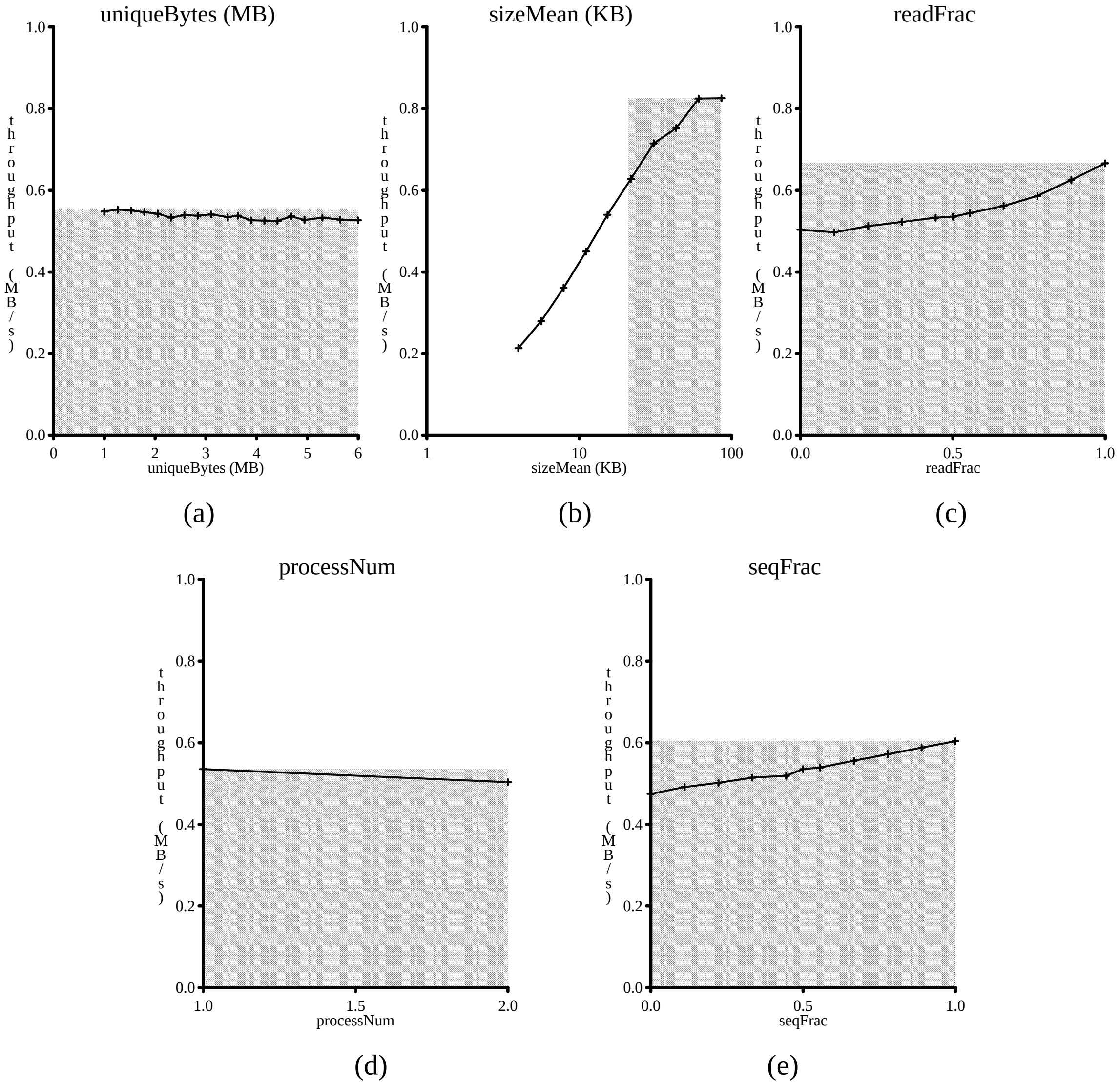

Figure 4.13: Self-Scaling Results using the Raw Disk Interface. This figure shows the results from a self-scaling benchmark of a SPARCstation 1+ using the raw disk interface. The focal point chosen for this interface by the self-scaling benchmark is uniqueBytes 3.5 MB, sizeMean = 15 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. The file cache is bypassed when performing I/O through the raw disk interface, hence the uniqueBytes curve is flat in graph (a). Reads are still faster than writes due to the prefetching track buffer on the CDC disk which benefits reads more than writes.

- The file cache is quite large, about 300 MB (Figure 4.12e), which is consistent with a main memory size of 384 MB.
- When accessing the file cache, writes are drastically slower than reads (Figure 4.12b). It appears that the Solbourne file cache uses a writing policy, possibly write-through, that

causes writes to the file cache to perform at disk speeds. Because writes derive essentially no benefit from the file cache, performance when varying uniqueBytes changes more gradually than it does with the other systems.

4.5.1.6. Raw Disk Interface

Some applications, such as databases, avoid going through the normal file system interface. Instead, they access the disk directly through the *raw disk interface* [Leffler89]. Figure 4.13 shows benchmark results for the SPARCstation 1+ using the raw disk interface (all other runs have been done using the file system, or *cooked* interface). The main difference of the raw disk interface is its bypassing of the file cache. Hence, the uniqueBytes curve is flat in graph (a). Reads are still faster than writes due to the prefetching track buffer on the CDC disk which benefits reads more than writes. Because the raw disk interface does not go through the file cache, sequentiality improves performance, even at the small uniqueBytes focal point of 3.5 MB.

4.5.1.7. Client-Server

To this point, the self-scaling benchmark has measured the I/O performance of file servers, that is, computers with disks directly attached to them. Many environments instead have client systems connected to a file server via a network. This section gives preliminary results from running the self-scaling benchmark on two different client-server configurations (Figure 4.14). The configuration used in Figure 4.14a is an HP 720 client workstation accessing an HP 730 (same as Table 2.1) file server running DUX (HP's Distributed Unix protocol). The configuration used in Figure 4.14b is a SPARCstation 1+ client workstation accessing a separate SPARCstation 1+ file server running Sun's NFS protocol. Both configurations use an ethernet network to connect the client and server. The main reason the HP configuration gets much higher performance is that the DUX protocol allows clients to cache both reads and writes,

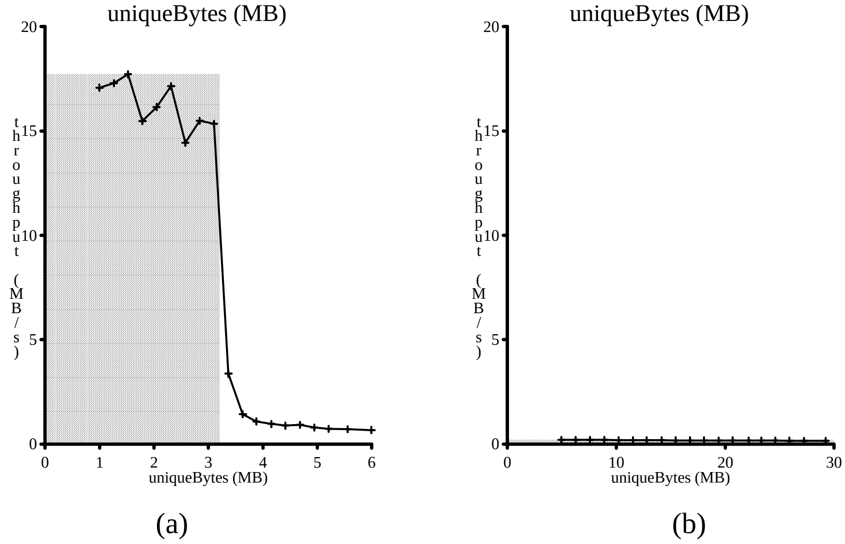

Figure 4.14: Self-Scaling Results for Two Client-Server Configurations. This figure shows the results from a self-scaling benchmark of two client-server configurations. The configuration used in graph (a) is an HP 720 client workstation accessing an HP 730 (same as Table 2.1) file server running DUX (HP's Distributed Unix protocol). The configuration used in graph (b) is a SPARCstation 1+ client workstation accessing a separate SPARCstation 1+ file server running Sun's NFS protocol. Both configurations use an ethernet network to connect the client and server. The main reason the HP configuration gets much higher performance is that the DUX protocol allows clients to cache both reads and writes, allowing I/Os to proceed at memory speeds rather than disk or network speeds.

allowing I/Os to proceed at memory speeds rather than disk or network speeds. The disadvantage of the DUX protocol is that reliability suffers because written data stays on the client for 30 seconds before stored on the server.

4.5.1.8. Unannounced Workstation

Figure 4.15 shows preliminary results from running the self-scaling benchmark on a beta release test version of an unannounced workstation[3]. Figure 4.15 reveals several unexpected pieces of information. First, Figure 4.15b shows that, in the file cache region, performance of

[3]This system was running a untuned, beta version of its operating system on pre-production hardware, so performance will likely increase dramatically over the next few months.

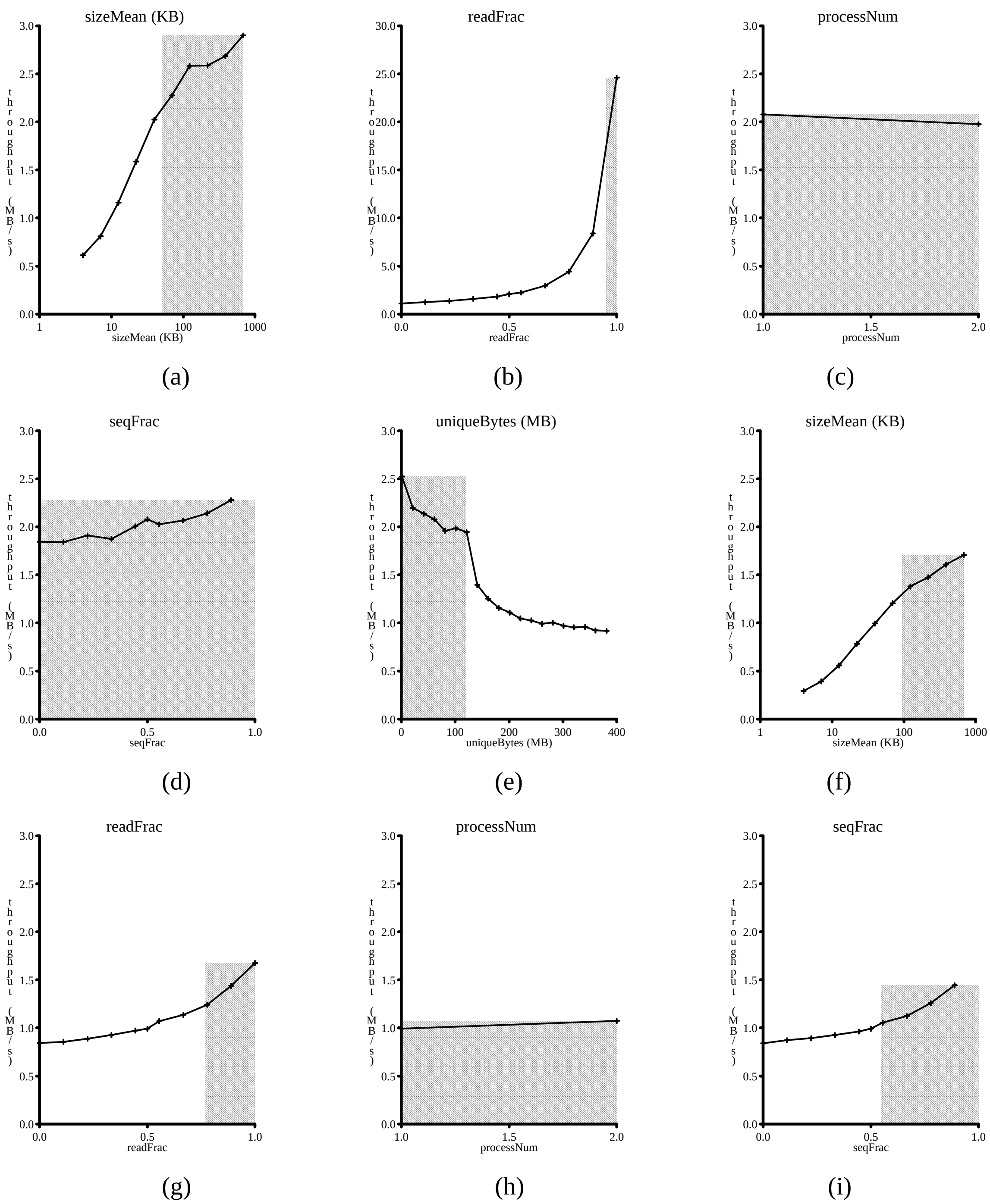

Figure 4.15: Self-scaling benchmark for unannounced workstation (beta release). This figure shows the results from the revised, self-scaling benchmark of a beta release of an unannounced workstation. The focal point for uniqueBytes is 61 MB in graphs a-d and 261 MB in graphs f-i. For all graphs, the focal points for the other parameters are sizeMean = 40 KB, readFrac = 0.5, processNum = 1, seqFrac = 0.5. Note how much slower writes are than reads when the file cache is used (graph b). Apparently, the cache write-back policy causes writes to perform at disk speeds even when all data fits in the file cache. Note that the scale of the y-axis is a factor of 10 larger in graph (b).

reads is an order of magnitude faster than writes. Like the Solbourne, writes are apparently being written-through the cache directly onto disk. Seconds, because of the write-through behavior, workloads with any writes get very disappointing performance. Overall system performance is no better than the last generation of workstations. Even with 100% reads, performance is no better than the HP 730. Unlike HP/UX, however, this operating system allows the file cache full use of the 192 MB of main memory.

4.5.2. Running Time

One disadvantage of the self-scaling benchmark is its running time. This is because each performance graph requires approximately 10 points with which to trace a fairly smooth curve. Moreover, with two distinct performance regions, the self-scaling benchmark reports 9 graphs, totaling 90 points. Each point can take 5-10 minutes to measure, due to multiple measurements for tight confidence intervals [MacDougall], which means that the whole process can take half a day or more. On the plus side, the self-scaling benchmark requires no human input and helps the evaluator gain much more insight into system behavior than do traditional, single point benchmarks.

As the capacity of main memory continue to quadruple every 2-3 years [Moore75], file caches will continue to grow larger; hence, larger values of uniqueBytes will be necessary to fill the file cache and stress the underlying disk system. As uniqueBytes increases, the running time of I/O benchmarks that try to stress the disks system will increase proportionally. I can see no way to avoid this linear increase in the running time of any I/O benchmark that goes through the file cache, since each workload measured by Willy will experience this slowdown.

Real applications will, of course, benefit from the trend toward larger memories by needing to go to disk less and less frequently. When I/O benchmarks that fill the cache take too long to run, real applications will take even longer to fill the cache and users will either not care about I/O performance in the steady state or will be willing to use long-running I/O benchmarks.

4.6. Conclusions

I have shown how a self-scaling benchmark can help an evaluator understand I/O system performance. For example, the evaluations in this chapter provide information on appropriate blocksizes, effective file cache sizes for reads and writes, the usefulness of higher concurrency, cache write policies, and absolute throughput. The self-scaling evaluation also met many of the other goals detailed in Section 2.2. It provides an effective, scaling strategy by scaling aspects of the workload to the changing performance characteristics of systems, projecting that users will tend to purchase systems that perform well on their applications. The self-scaling benchmark can be especially useful to the systems programmer trying to understand the performance implications of the I/O code that he or she is writing. For instance, the programmer can develop the code, run the self-scaling benchmark overnight, then see the range of workloads on which the code performs well.

As intended, performance of the benchmark is I/O limited—almost all workloads run here spent more than 90% of their time performing I/O. The self-scaling benchmark is also applicable to a wide range of audiences—by graphing performance curves for each parameter, more applications have their workload included in the benchmark.

However, the self-scaling benchmark does not yet completely meet the goal of meeting the performance evaluation needs for all audiences; there are still many workloads not included in the results of the self-scaling benchmark. Also, self-scaling benchmarks make it difficult to compare two machines directly, because the benchmark will select a different focal vector for each machine. The next chapter addresses these shortcomings by predicting performance for any workload.

4.7. References

[Convex89] *C Series Data Sheet—Input/Output Subsystems*, CONVEX Computer Corporation, 1989.

[Convex90] *The First Family of Open Supercomputing*, CONVEX Computer Corporation, 1990.

[Convex91] *C3200*, CONVEX Computer Corporation, 1991.

[Hong91] W. Hong and M. Stonebraker, ‘‘Optimization of Parallel Query Execution Plans in XPRS’’, *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, Miami, FL, December 1991. also to appear in Journal of Distributed and Parallel Databases.

[Horning91] R. Horning, L. Johnson, L. Thayer, D. Li, V. Meier, C. Dowdell and D. Roberts, ‘‘System Design for a Low Cost PA-RISC Desktop Workstation’’, *Procedures of the IEEE Computer Society International Conference (COMPCON)*, Spring 1991, 208-213.

[IDC91] *Integrated Disk Channel (IDC)*, CONVEX Computer Corporation, 1991.

[Kim87] M. Y. Kim, A. Nigam, G. Paul and R. J. Flynn, ‘‘Disk Interleaving and Very Large Fast Fourier Transforms’’, *International Journal of Supercomputer Applications 1*, 3 (Fall 1987), 75-96.

[Lee92] E. K. Lee, P. M. Chen, J. H. Hartman, A. L. C. Drapeau, E. L. Miller, R. H. Katz, G. A. Gibson and D. A. Patterson, ‘‘RAID-II: A Scalable Storage Architecture for High-Bandwidth Network File Service’’, UCB/Computer Science Dpt. 92/672, University of California at Berkeley, February 1992.

[Leffler89] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison-Wesley Publishing Company, 1989.

[MacDougall] M. H. MacDougall, *Simulating Computer Systems, Techniques and Tools*, The MIT Press. Computer Systems Series.

[Moore75] G. E. Moore, ‘‘Progress in Digital Integrated Electronics’’, *IEEE Digest International Electron Devices Meeting*, 1975, 11.

[Ousterhout85] J. K. Ousterhout, H. Da Costa and *et al.*, ‘‘A Trace-Driven Analysis of the UNIX 4.2 BSD File System’’, *Operating Systems Review 19*, 5 (December 1985), 15-24. Proceedings of the 10th Symp. on Operating System Principles.

[Rosenblum92] M. Rosenblum, Sprite LFS Write Cache Size, personal communication, July 6, 1992.

[SPEC91] *SPEC SDM Release 1.0 Manual*, System Performance Evaluation Cooperative, 1991.

[TPCB90] *TPC Benchmark B Standard Specification*, Transaction Processing Performance Council, August. 23, 1990.