SCS&E Report 9312
November, 1993

Address Space Management Issues in the Mungi Operating System

Kevin Elphinstone

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES

Abstract

The Mungi operating system features a single 64 bit persistent address space encompassing all data in the system. This differs dramatically from current generation operating systems in which each process has its own address space and persistent data is stored in a filesystem.

This report is a preliminary investigation of address space management issues raised by adopting a single persistent address space model. Issues examined are internal and external fragmentation of the address space, reuse versus no-reuse allocation policies, and page table structures used to support the address space.

1 Introduction

Mungi is a single address space operating system supporting persistence[2, 6]. It features a single 64 bit namespace encompassing all data contained in the system. Mungi does not have a traditional file system; instead, Mungi relies on distributed persistent shared memory for storage.

Distributed persistent shared memory is similar to distributed shared memory systems, with the addition of persistent storage (disks) to both increase the amount of shared memory available via virtual memory techniques and provide persistent storage of data in the address space. Thus traditional files on disk are replaced by objects in the distributed persistent address space.

A name service is provided to map names of objects to addresses of objects so the user's view of objects is similar to files in that they can be referred to and grouped under meaningful names. Note that users are free to implement their own name service, as it is really the addresses that name objects, and users may choose to store addresses of their objects in any way they choose.

2 Internal Fragmentation

In the Mungi single address space operating system the traditional file system no longer exists. In its place is a persistent object space that merges the files system into the single address space. The traditional view of a file being a stream of data read from a disk is no longer valid. The abstraction of an object in Mungi is a region of memory that can be accessed at random like other virtual memory.

Current processor architectures provide page based memory management so detection of how much of an object is used, and which part of an object in the case of sparse objects can only be done with page granularity. Protection of memory is also page based. These limitations of current architectures argues strongly for page alignment of objects and rounding of objects to the nearest multiple of a page to simplify protection and backing store management.

When storage requests are rounded up to the nearest multiple of some storage allocation unit, then the storage wasted by doing so is referred to as internal fragmentation. The effect of this fragmentation on overall storage required and on individual files can be estimated by a static analysis on a typical object distribution.

The object size distribution analysed is illustrated in Figure 1. The distribution's source is the actual distribution of 240000 files found on our local network of UNIX workstations. The distribution is very similar to distributions found in other file system studies[1, 5]. It can be argued that a typical file distribution may not be a typical object distribution. However data on typical object distributions for single address space systems is not yet available, and given that a typical user's file usage patterns will not change instantaneously, current file distributions should suffice for experimental analysis.

2.1 Storage Requirement

Different architectures support different page sizes. Internal fragmentation will vary for different pages sizes as will overall storage requirements. Overall storage requirements were

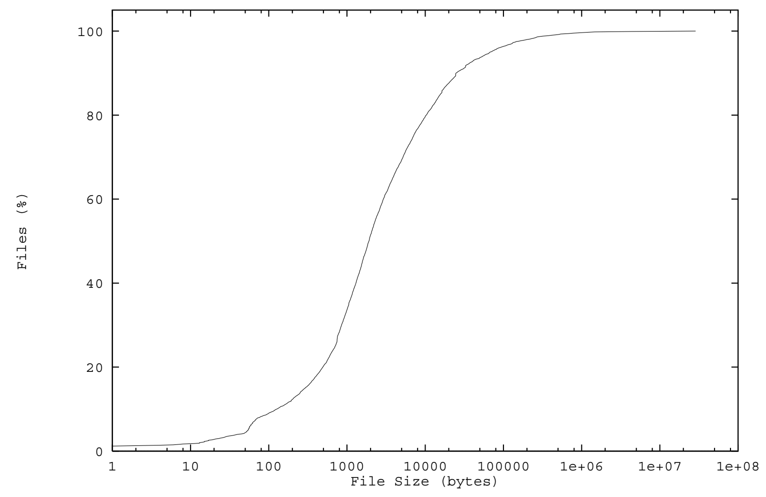

Figure 1: Cumulative distribution of file sizes.

analysed for different page sizes to gauge the effect page size has on storage. Various allocations unit sizes from 1 byte (the best case) to 1 Mbyte were tested with the typical file distribution. The results in Table 1 shows the percentage extra storage required to support each of the allocation unit sizes.

It can be seen that internal fragmentation is not significant in relation to overall storage required until the unit size approaches 1 Kbyte, above which it soon begins to get very expensive. This limits the choice of a suitable allocation unit size for the given distribution to around 4 Kbytes and below, unless multiple allocation unit sizes can be supported, then the smaller size should preferably be around 2 Kbytes or less.

2.2 Summary

These results are dependent on the file distribution under study. A significantly different distribution will change the numbers illustrated, though it is encouraging that from studies published it seems that the typical distribution is indeed fairly typical.

However one can intuit that an increase in the proportion of smaller files will result in an increase in overall storage requirements together with an increase in the proportion of files wasting space and thus poorer results for a given allocation unit size. An increase in the proportion of larger files will improve the results illustrated.

Access patterns are not taken into account here. However, if access patterns significantly favour smaller files, then the load on system resources such as physical RAM, network bandwidth and disk bandwidth will increase.

To summarise, efficient fine grain file (or object) support requires a fine grain allocation unit size. To use the typical virtual memory page size of 4 Kbytes will require system structure and policies that encourage and support larger grain objects. Fine grain object support (finer than the

Alloc. Unit	% Extra Storage
1	0
2	0.002
4	0.005
8	0.013
16	0.027
32	0.056
64	0.113
128	0.233
256	0.478
512	0.988
1024	2.009
2048	4.256
4096	9.516
8192	21.19
16384	46.67
32768	100.3
65536	212.5
131072	441.5
262144	906.9
524288	1848
1048576	3737

Table 1: Storage allocation units and corresponding percentage extra storage required to support the allocation unit size.

typical distribution illustrated) will need to be supported by higher level software by grouping small objects within a large grain one using an arbitrary data structure.

3 External Fragmentation

Regions of memory in dynamic allocation systems can remain unused as they are too small to fulfil storage requests. This phenomenon is termed external fragmentation. External fragmention in Mungi as a result of inter-object gaps in the address space does not result directly in wasted physical memory or storage as available virtual memory regions are not held in memory nor backed to disk until they are allocated. However external fragmention does indirectly effect page table population density, and thus in a multilevel tree page table it effects the overall page table size for a given number of allocated pages.

Memory in Mungi is persistent and so is external fragmentation. External fragmentation cannot be aleviated by garbage collection and compaction of the address space. Moving objects in the address space changes the objects name and potentially corrupts the internal object structure if it contains pointers. System restart results in the same address space layout that existed at system shutdown. Minimisation of external fragmentation can only achieved by allocation policy, not by post allocation processing.

Knuth[4, pages 445–451] derived the *Fifty Percent Rule* which attempts to predict the ratio of the equilibrium number of free blocks in relation to the equilibrium number allocated blocks (p). The rule states the $p \approx \frac{1}{2}p_f$, where p_f is the probability of any given allocation not exactly filling the free block. Typically allocation requests never exactly fill a free block and thus p_f is 1, and the number of free blocks is approximately half the number of allocated blocks.

The situation should improve in Mungi as allocation requests are rounded up to the nearest multiple of some allocation unit and the free blocks are also multiples of the same allocation unit. The probability of a perfect fit in this situation should be better at the expense of increased internal fragmentation. The question is how does the page size effect the probability of a fit and external fragmention, and are the sizes that are appropriate to reasonable levels of internal fragmentation also suitable for minimal external fragmention. A simulation was undertaken to investigate the phenomena.

3.1 Simulation Description

The simulation consists of generating files using the distribution in Figure 1, and each file generated survives a lifetime given by the lifetime distribution illustrated in Figure 2. The lifetime distribution is taken directly from Baker's file system studies[1] and is not correlated with the file size distribution. The simulator generates files at a rate of approximately 350 Mbytes per simulated day; a statistic that is similar to published studies.

Each file is allocated in the single address space, followed by deallocating them after their lifetime has expired. Files are allocated using the *first fit* algorithm[4, pages 437-438] and various page sizes are used to assess their effect on p_f and external fragmentation.

The simulation was run for a simulated month. The following variables were sampled at intervals of 10 minutes simulated time.

Free blocks The number of free blocks of memory.

Allocated Blocks The number of allocated blocks of memory.

Allocations The number of allocations made so far in the simulation.

Links Searched The number of free blocks scanned before a block of the correct size is found.

Perfect Fits The number of allocations that perfectly fit the free block allocated.

3.2 Simulation Results

Figure 3 shows how the ratio of free blocks to allocated blocks varies for different page sizes. The figure reveals the Knuth's fifty percent rule approximately holds for the 1,2,4,8 byte case illustrated at the top of the graph. However as the allocation unit size increases and correspondingly the percentage of files less or approximately equal to the allocation unit size increases,

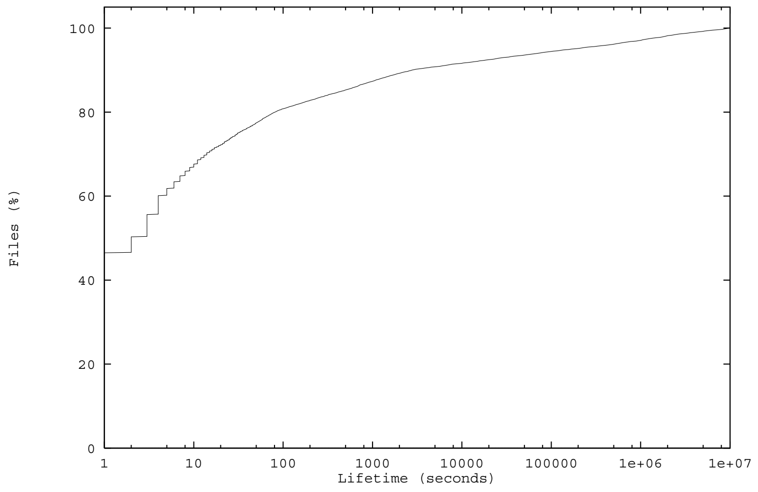

Figure 2: Cumulative distribution of file lifetime.

then the ratio of free blocks to allocated ones decreases. For the case of a 4 kilobyte page, the number of free blocks is approximately 3% of the number of allocated blocks.

This dramatic decrease in external fragmentation with respect to allocation unit size can be explained by referring to Figure 4. Knuth predicts the ratio of free to allocated will be approximately 50% of the probability of allocation not fitting the free block. The figure shows a dramatic drop in the probability of an allocation not fitting as the page size is increased. This translates to the decrease in the number of free blocks.

The decrease in the number of free blocks has a beneficial side effect on the performance of the first fit algorithm. Figure 5 illustrates the dramatic drop in the number of free blocks needed to be searched for a block of sufficient size to be found, as the page size is increased. The number of allocated blocks remains constant at approximately 12000 at the end of each simulation run for each page size.

3.3 Summary

The simulations show that the allocation unit size has a dramatic effect on external fragmention. While increasing the allocation unit size had a detrimental effect on internal fragmentation, it is beneficial for external fragmentation. The implies a trade-off when selecting a suitable allocation unit size for Mungi. The typical page size of 4 kilobytes available on most current hardware architectures fortunately appears to be a good initial compromise, and without detailed dynamic performance benchmarking, tuning the allocation unit size would be foolish.

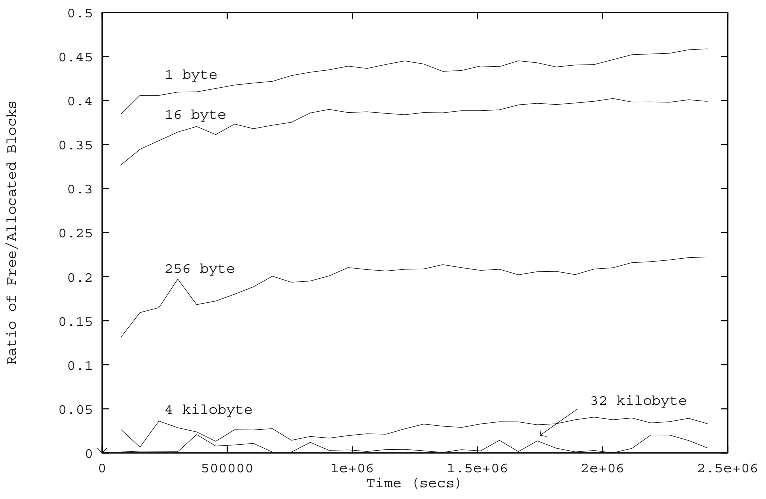

Figure 3: Ratio between the number of free and allocated blocks over time for various page sizes.

4 Reuse versus No Reuse

Simulations in section 3 were carried out using a first fit algorithm of memory allocation. First fit can supposedly be improved by searching for a free block from the place it was last allocated from. This is intended to prevent the clustering of a large number of small unusable free blocks at the beginning of the address space, thus decreasing the search time to find a usable free block. With a 64 bit address space this policy would be reduced to a policy of effectively never reusing address space as the persistent system MONADS[3] does.

Never reusing the address space will effect the amount of address space spanned by the amount of data in it, and hence will effect the population density of the page table and size. Simulations were carried out to investigate the effect of reuse versus no reuse on page table size.

4.1 Simulation Description

The simulation was carried out using a 6-level tree page table indexed as illustrated in figure 6. In section 3.1, files were generated, allocated, and de-allocated. To compare reuse to no reuse, the same files were allocated into two address spaces with separate page tables, one address space using a no reuse policy and one address space using first fit. The page size for the simulation was 4 kilobytes.

4.2 Simulation Results

The simulation was run for 3 months simulation time generating 40 gigabytes of data of which 500 megabytes was still allocated at the end of the simulation time. Figure 7 illustrates how the

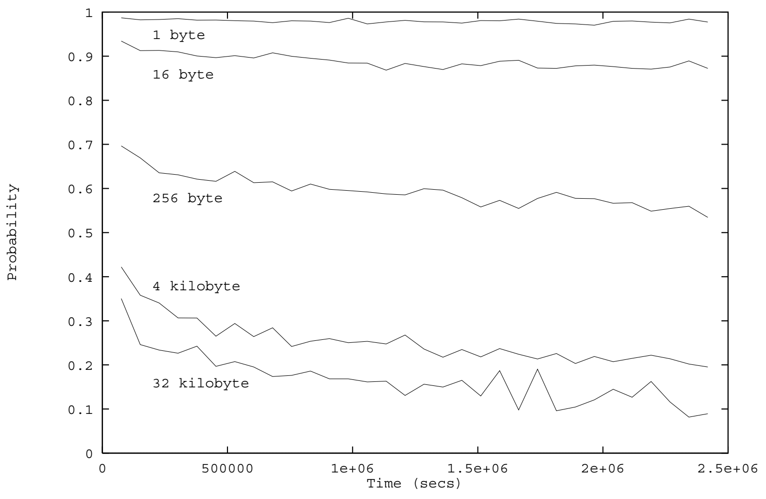

Figure 4: Probability of a given allocation not perfectly fitting the free block size, for various page sizes.

page table size varies during the life of the simulation.

At the end of the simulation the first fit policy produced a page table size of 1.2 megabytes which represents 0.2% of the data the page table maps. For the no reuse case the page table was 35 megabytes which is 7% of the size of the data mapped, an increase of 30 times the page table size for the first fitted address space.

The reason for the dramatic difference in page table size is the difference in the population density of the tables. Both tables map the same amount of data, however the span of the tables vary greatly. The upper bound of pages mapped obviously tracks the amount of data generated for the no reuse case, where as the upper bound for the first fit case tracks only 10% above the amount of data mapped. Hence for the no reuse case, the page table size will increase with time. As can be seen from Figure 8, the data is concentrated in the upper region of the address space with a long tail of older allocated data leading back to the beginning of the address space. This situation will not change much as time goes by, except that the tail will get longer and thus page table size worsen.

The first fit case stored the data in a comparatively small region at the beginning of the address space, and hence has a much smaller page table. The efficiency of the page table as time continues will depend on external fragmentation of the data, but this is small due to the 4 kilobyte allocation unit size, and will never be as bad as the no reuse case.

4.3 Summary

A multilevel tree structure efficiently supports page table type mappings only in co-operation with a suitable allocation policy. Multilevel trees work best when the mapping density is high in allocated regions of the address space and the first fit policy does provide a high density of

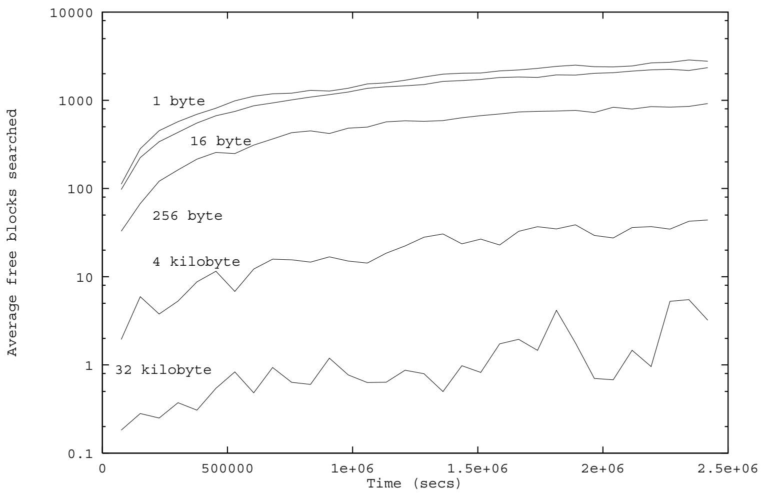

Figure 5: The average number of free blocks searched before a block of sufficient size is found, for various page sizes.

Level 1	2	3	4	5	6	
63−57	56−48	47−39	38−30	29−21	20−12	11−0

Figure 6: The division of bits for indexing the various levels of the page table.

mappings.

A no reuse policy does not provide a dense mapping of data and thus is not suited to use with a multilevel tree page table. If a no reuse policy was required, other data structures would be needed, probably based on hashing. However it is unlikely that it would be significantly better than a multilevel tree if the mappings are as densely populated as they are using first fit policy in the above situation.

5 Page Table Indexing

Mungi divides the address space up into partitions managed by the different machines on the network[2]. The upper nine bits[1] of an address is used to indicate the address space partition an object was allocated from, and is termed the *address-space partition identifier* (API). This leaves 36000 terabytes of address space per partition. It is unlikely that a machine will have anywhere

[1] Nine bits was chosen for this example, however it will be closer to 10 or 12 bits in the real system.

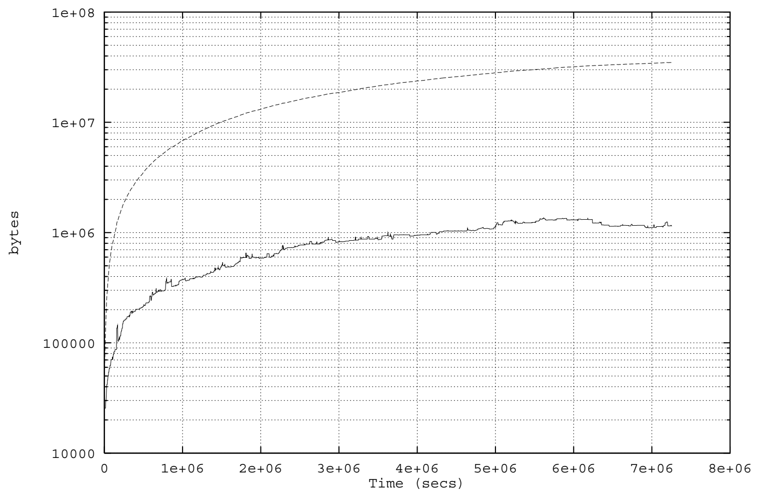

Figure 7: The page table size over the simulation time for no resue (top) and first fit (bottom) allocation policies.

near this amount of physical storage so an address space partition will be mostly empty due to the limited amount of storage a node can physically provide.

Each machine allocates objects out of its own partitions which leads to the situation where the overall address space consists of large memory partitions with objects allocated starting from the beginning of each partition, and moving towards the end. Each machine will have it own page table that maps local data, and remote data recently referenced locally. This scenario gives rise to a possible optimisation of the page table by modifying the indexing of the 6-level tree.

The optimisation relies on the fact that the middle bits of an address of a page are less actively used than the upper (API) bits. If the middle bits are used to index higher levels in the tree, then the tree should have reduced fan-out for a given amount of storage allocated. The API of an address changes depending on the origin of the object the address refers to. In an actively mobile distributed system the API bits will not tend to a particular constant, and hence should be used to index lower levels in the tree to reduce fan-out.

5.1 Simulation Description

To see the effect of the optimisation a simulation was carried out comparing the indexing arrangement of section 4.1 with an optimised arrangement. For the case study partitions were only active for approximately the first 600 megabytes. This number corresponds approximately to the lower 30 bits of the address space. These bits were used to index the lowest levels of the page table. The bits above 30 to the API were inactive and thus used to index the top levels of the page table, leaving the API bits to index the middle level. Figure 9 illustrates the optimised indexing arrangement.

Two network configurations were tried, one with 32 nodes and one with 256 nodes. The ratio

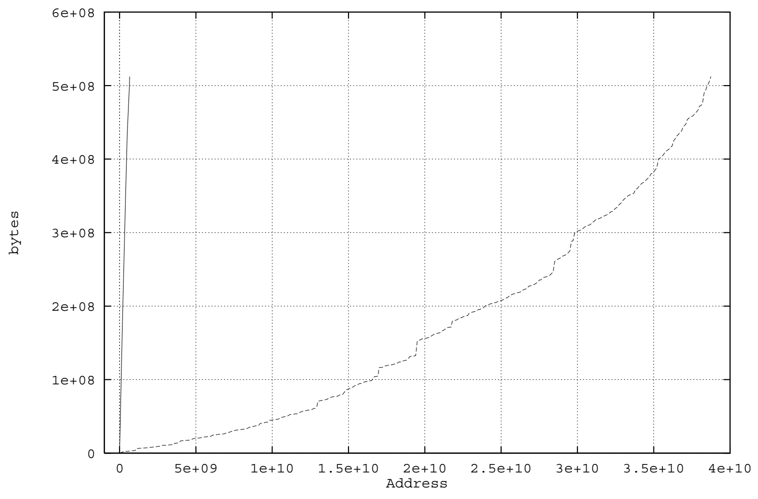

Figure 8: The cumulative spread of data in the address space for first fit (solid line) and no reuse (broken line).

of remote to local mappings was varied between 0% and 100% in increments of 10%, where 0% represents the case where all the data is under the single local API, and 100% represents the case where all the data was spread evenly between all remote node APIs.

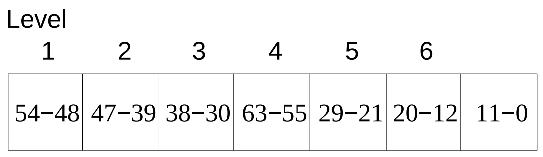

Figure 9: Optimised page table indexing.

The two page tables were built for each configuration which mapped the same amount of data which was approximately 600 megabytes.

5.2 Simulation Results

The results of the simulations are shown in Figure 10. The page table grows in all four cases as the remoteness of data increases, as remoteness effectively increases the span in the address space of the data being mapped. The normally indexed page table is larger than the optimised one for both the 32 node and 256 node case. The optimised page table was the same size as the normal one for the case of 0% remote data mapped. However for 100% remote data, the optimised page table was 2.0% smaller than the normal page table for the 32 node case and 3.1% smaller for the 256 node case. This marginal improvement is the best that can be expected for the case studied as the “busiest” bits are all indexing the lowest levels of the tree. In a sparser and regular address space layout a greater improvement would be expected, however we do not believe this will be the case in Mungi.

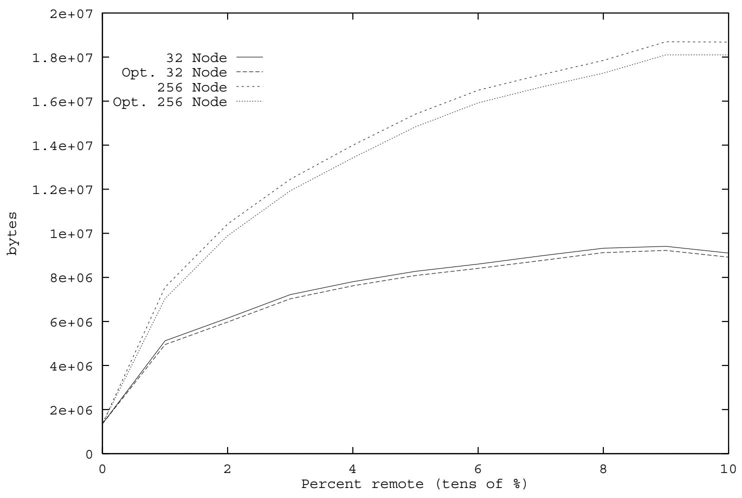

Figure 10: Graph of page table size versus percentage remotely mapped data, for normal and optimised indexing in the 32 node and 256 node cases.

5.3 Summary

As discussed in section 4.3, multilevel tree page tables work best when heavily populated. If a particular policy allocates so as spread data in the address space in a regular manner, then the indexing of the levels in the page table can be changed to achieve a denser population. A denser tree is achieved by indexing higher levels of the tree with relatively constant bits and lower levels of the tree with highly random ones.

6 Conclusion

Basic issues of address space management in Mungi have been examined. Preliminary investigation indicates that internal fragmentation should not be a problem on current architectures. External fragmentation of the address space is controllable provided a reuse of address space policy is used. This leads to a densely populated address space and hence densely populated multi-level tree page tables. Multi-level tree page tables can be further compacted by using indexing that suits the address space layout.

7 Further Work

Multi-level page tables are not efficient if the address space is sparsely populated. Sparse address spaces can be discouraged via allocation policy, but not prevented. Hash based page tables need to be investigated to determine suitability for both sparse and densely populated address spaces.

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurement of a Distributed File System. *13th ACM Symposium on Operating Systems Principles*, October 1991.

[2] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Graham R. Hellestrand. A Distributed Single Address-Space Operating System Supporting Persistence. Technical Report 9302, School of Computer Science and Engineering, The University of New South Wales, March 1993.

[3] Frans Alexander Henskens. A Capability-Based Persistent Distributed Shared Memory. Technical Report 462, Basser Department of Computer Science, University of Sydney, Australia, March 1993.

[4] Donald Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[5] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. *10th ACM Symposium on Operating Systems Principles*, 1985.

[6] Stephen Russell, Alan Skea, Kevin Elphinstone, Gernot Heiser, Keith Burston, Ian Gorton, and Graham Hellestrand. Distribution + Persistence = Global Virtual Memory. In *Int'l Workshop on Object-Orientation in Operating Systems*, volume 2, pages 96–99, Dourdan, France, 1992. IEEE.