Tiled Virtual Memory for UNIX

James Franklin – Kodak Electronic Printing Systems

ABSTRACT

Many computer applications require the manipulation of large data arrays. These applications can behave badly under a paged virtual memory (VM) system, due to poor memory access patterns. One solution to this problem is tiling, a technique in which the arrays are partitioned into sub-arrays that map one-to-one with VM pages. Software implementations of tiling have been described in the literature, but none provide the speed and application transparency of a hardware solution.

We have implemented a hardware based, tiled VM within a version of the UNIX operating system. Based on a novel memory management unit and supporting kernel software, this tiled VM has proven to be an efficient environment for manipulating 2-dimensional arrays of data.

In this paper we discuss the kernel changes required to implement our tiled VM. We then compare tiled and paged versions of our VM system, and show that tiling results in a 50-fold reduction in working set size for a common class of image processing algorithms.

Introduction

There are many computer applications that require the manipulation of large, multi-dimensional arrays. Example applications include image processing, graphics rendering, numerical analysis involving large matrices, and simulation of physical systems. These applications can behave poorly under a conventional VM system, due to memory access patterns that result in large working sets[1] [6][4][1][9].

Applications that manipulate large arrays can benefit greatly from *tiling*, a technique in which large data arrays are partitioned into a number of identically sized sub-arrays, and the sub-arrays mapped (via software or hardware) to the underlying virtual pages. Algorithms for manipulating these tiled arrays often show dramatic reductions in working set size, resulting in less paging activity and faster execution times.

In a pioneering paper, McKellar and Coffman [6] investigate the performance of various matrix algorithms in a paging environment. They conclude that the use of sub-arrays can improve paging performance by orders of magnitude. Blinn [1] discusses the advantage of tiling in a graphics rendering application, and reports a 10-fold reduction in the number of page faults. Wada [9] presents a software implementation of tiling for image processing, and compares various tile replacement and prefetch algorithms. Another software implementation of tiling for image processing is described by Ryman [7]. But none of these implementations take advantage of the speed and application transparency that a hardware solution offers.

We have implemented a hardware based, tiled virtual memory within SUNOS, Sun Microsystems' implementation of the UNIX operating system. This tiled VM is based on a custom memory management unit called the IMMU and supporting kernel software. Together, they provide a tiled, shareable, virtual memory that we call *image memory*. As the name implies, we have used this tiled VM for image processing applications, although it would be equally useful for any of the applications mentioned above.

The IMMU and the tiled VM system described in this paper are currently being used by customers in a mid-range, color electronic prepress system called the Kodak Prophecy Color Publishing System.

In the next section we examine the motivation for using tiling techniques for large arrays. This is followed by an overview of the IMMU and the virtual to physical address translation in *IMMU Hardware*. In *Image Memory Concepts* we describe our implementation of image memory and the new system calls that support it. Tile fault handling, tile-in and tile-out, and new kernel daemons are described in *Kernel Software*. This is followed by a comparison of a tiled and paged version of our VM system in *VM Performance*. Finally, we close with a discussion of future work.

Tiling versus Paging

The fundamental benefit of tiling over paging is that processes that manipulate tiled arrays often show dramatic reductions in working set size. To illustrate this effect, we consider the manipulation of large images. Figure 1 shows a row-ordered, 2-dimensional array representing an image. The array has 3K rows containing 4K pixels per row, with each pixel occupying one byte. We want to manipulate this array on a machine that has 8 megabytes of physical memory available for paging, and a virtual page size of 4 kilobytes.

Now, consider a process that manipulates the array in paged virtual memory. In process virtual space, the array lies in a contiguous range of virtual addresses. Assuming that the first pixel is page aligned, each row fills one 4 kilobyte page (Figure

[1] Denning [2] defines the working set of a process to be the set of pages referenced by that process in some time interval of interest.

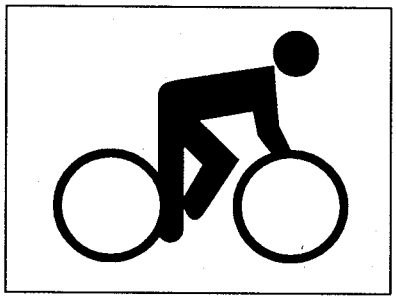


Figure 1: Row-Ordered Array in Virtual Memory

2). If the process accesses the array in row order, the array will be paged in and out of memory in an efficient fashion. But if the algorithm requires the process to access the array by columns, the process will suffer a page fault for every pixel accessed[2].

On the other hand, suppose the same process manipulates the same array in tiled virtual memory, using 256x256 byte tiles. In process virtual space, the array still lies in a contiguous range of virtual addresses. However, that range of addresses is now mapped to virtual tiles (Figure 3). If V is the virtual address of the first byte in the array, and V is mapped to physical tile T, then the virtual bytes at V, $V+1$, $\cdots$, $V+255$ are mapped to sequential physical bytes at T, $T+1$, $\cdots$, $T+255$. The virtual byte at $V+256$ is mapped to a different physical tile T'. The virtual byte at $V+4096$ (the first byte of the second array row) is again mapped to physical tile T, at $T+256$.

With tiled virtual memory, row and column access are equally efficient. If the array is accessed by columns, e.g., the process will suffer a tile fault every 256 pixels in the first column, and then run without faults for the next 255 columns.

Note the difference in working set size during column access to the paged and tiled arrays. For a column of the paged array, the working set size is 3K pages (the entire array), or 12 megabytes. For a column of the tiled array, the working set size is just 12 tiles or 786 kilobytes.

Tiled virtual memory can also reduce the working set size for many localized array operations. For example, consider an application that performs a local editing operation on the bicycle rider's shirt, such as a color or texture change. In paged memory (Figure 4), the working set size is approximately 1024 pages or 4 megabytes. In tiled memory (Figure 5), the working set size is 19 tiles or 1.2 megabytes.

[2]A column of pixels touches every virtual page in the array, requiring 12 megabytes of physical memory per column. The host machine has only 8 megabytes of physical memory available. Assuming a conventional LRU (Least Recently Used) replacement algorithm, linear passes through the array columns will cause the entire array to be repeatedly paged in and out of physical memory.

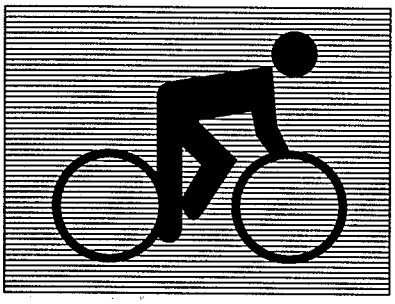


Figure 2: Mapping of Array to Virtual Pages

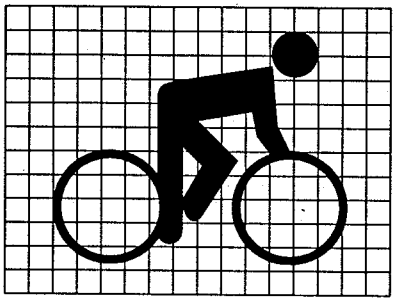


Figure 3: Mapping of Array to Virtual Tiles

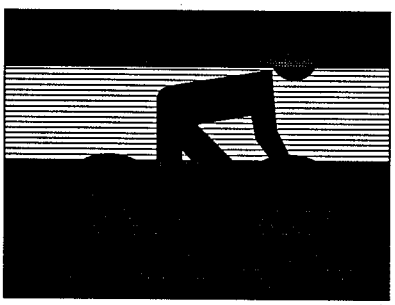


Figure 4: Pages Needed for Local Editing

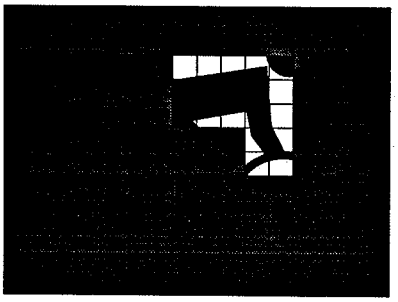


Figure 5: Tiles Needed for Local Editing

IMMU Hardware

Functional Overview

A functional overview of the IMMU hardware is shown in Figure 6. The IMMU consists of tile address mapping logic and a conventional MMU. Array virtual addresses from the CPU pass through the tile address mapping logic and are converted to tiled virtual addresses. These addresses then pass through the MMU and are converted to physical memory addresses via conventional means.

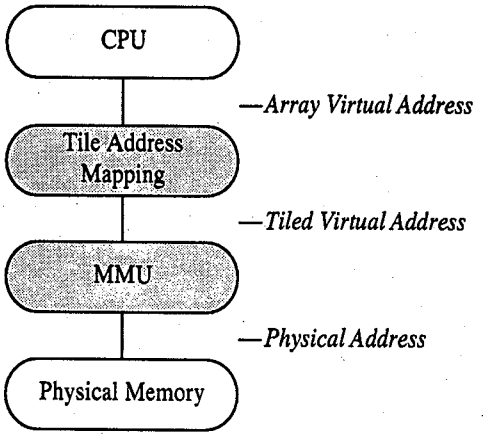


Figure 6: IMMU Functional Overview

The IMMU is located on the system bus, and responds to two address ranges: a 512 megabyte physical space, and a 1 gigabyte virtual space. All accesses to IMMU physical space are mapped directly to IMMU physical memory, which provides physical tiles to support the tiled virtual memory. Access to this physical space is restricted to supervisor mode, and is only done during tile-in, tile-out, and copy-on-write tile faults.

Access to IMMU virtual space invokes the IMMU virtual to physical mapping, and the access is redirected to the appropriate physical tile. During an access to IMMU virtual space, a tile fault will occur if the corresponding tile table entry is marked as invalid or if the access violates the virtual tile's protection. A tile fault causes the IMMU to generate a bus error signal.

Tile Address Mapping

When IMMU virtual space is accessed, the IMMU must translate the array virtual address to a tiled virtual address, which is then passed to the MMU. Internal views of an array virtual address and a tiled virtual address are shown in Figure 7 and Figure 8, respectively. An array virtual address consists of a Y address and an X address portion. The length of the Y and X portions can vary from 8 to 16 bits, corresponding to image dimensions from 256 bytes to 64 kilobytes. During tile address mapping, the lower 8 bits of the Y and X addresses (Y_{LOWER} and X_{LOWER}) are extracted and combined to form a tile offset. The upper portions of the Y and X addresses (Y_{UPPER} and X_{UPPER}) are extracted and combined to form a virtual tile number. Finally, the tile number and tile offset are concatenated to form the tiled virtual address.

In order to extract Y_{LOWER} and X_{UPPER} from an array virtual address, the IMMU needs to know the array's XSIZE – the number of bits in the X address. During tile address mapping, the upper bits of the array virtual address are used to index into an XSIZE table to obtain the number of bits in the X address. This effectively breaks the IMMU virtual space into 256 segments of 4 megabytes each. All images in the same segment have the same XSIZE. Images longer than 4 megabytes occupy multiple segments.

Y Address		X Address	
Y_{UPPER}	Y_{LOWER}	X_{UPPER}	X_{LOWER}

Figure 7: Anatomy of an Array Virtual Address

Tile Number		Tile Offset	
Y_{UPPER}	X_{UPPER}	Y_{LOWER}	X_{LOWER}

Figure 8: Anatomy of a Tiled Virtual Address

Image Memory Concepts

Image Memory as Shared Virtual Memory

Image memory is treated as a type of shared virtual memory. Processes request the allocation of an image in image memory and specify its attributes, such as size, read/write protection, and access rights. The kernel returns to the requesting process a unique image identifier for the allocated image. This identifier is used to reference the image or to allow another process to share the same image, by passing the image identifier. A reference count is maintained for each allocated image, and an image is freed when the reference count drops to zero. The separate components of a color picture (such as (u,v,L), (R,G,B), or matte) are stored as separate images in image memory.

Image memory is tiled in 64 kilobyte chunks to and from the swap device. Swap space for an image is allocated when an image is allocated, and deallocated when the image is deallocated. This swap space is associated with the allocated images, not with the processes that created them. Since image swap space is associated with each image, processes

sharing an image therefore share the same image swap space.

New System Calls

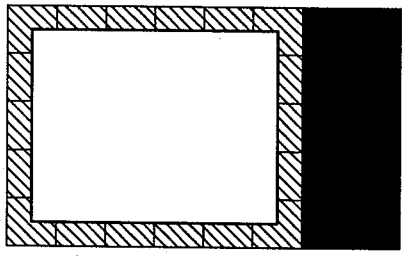
There are four new system calls to support tiled image memory. Image memory is allocated with *imem_alloc*, shared with *imem_share*, and freed with *imem_free*. The protection of an image can be changed with *imem_prot*.

imem_alloc

```
caddr_t
imem_alloc(x_dim, y_dim, image_area,
           access, fill_color)
int      x_dim, y_dim;
rect_t   *image_area;
access_e access;
int      fill_color;

typedef struct {
     int x_min, y_min, x_max,
         y_max;
} rect_t;

typedef enum {
        IMEM_PRIVATE, IMEM_SHARE
} access_e;
```



— Data pixels in *image_area* rectangle

— Unused pixels in partially used tiles

— Unused pixels in read-only border tiles

Figure 9: An Allocated Image

imem_alloc allocates a section of image memory large enough to hold an image of size *x_dim* by *y_dim*. The *image_area* rectangle defines the area of the image that the user process will be writing to. By selecting appropriate values for *image_area* and (*x_dim*, *y_dim*), it is possible to obtain a constant color border around the working image area. The image is private to the current process if access is IMEM_PRIVATE, and shareable between processes if access is IMEM_SHARE. If *fill_color* is in the range 0-255 then all pixels in the allocated image are set to *fill_color*. If *fill_color* is NO_COLOR then all pixels in the allocated image are left unchanged, and will have undefined values.

Due to hardware constraints imposed by the IMMU, the specified *x_dim* value is rounded up to the next power of 2, and *y_dim* is rounded up to the next multiple of 256 as image memory is allocated. As a result, there can be border tiles around the image that do not overlap with the *image_area* rectangle. Any such tiles have their protection set to read-only, no-fault on write (see Figure 9). Any tiles that overlap the *image_area* rectangle have their protection set to read-write. *imem_prot* can be used to change this protection.

imem_share

```
bool_t
imem_share(image)
caddr_t image;
```

imem_share requests that the current process be allowed to share the previous allocated *image*. The image must have been specified as shareable by the process that allocated it.

imem_free

```
bool_t
imem_free(image)
caddr_t image
```

imem_free informs the kernel that the specified *image* is no longer needed by the current process. If no other process is currently sharing the image, it will be deleted.

imem_prot

```
bool_t
imem_prot(image, image_area, prot)
caddr_t image;
rect_t  *image_area;
prot_e  prot;

typedef enum {
        IMEM_PROT_RO,
        IMEM_PROT_RONF,
        IMEM_PROT_RW
} prot_e;
bool_t
```

imem_prot changes the protection of the previously allocated *image*. The new protection can be read-only, read-only with no fault on write, or read-write. The image must have been allocated by the current process or have been specified as shareable by the process that allocated it. The protection of all tiles in the region of the image specified by *image_area* is altered according to the value of *prot*. Protection can only be altered on entire tiles, not individual pixels.

Constant Color Tiles

Multiple virtual tiles are often mapped to the same physical tile. Tile sharing is desirable because it saves physical tile memory, and allows an allo-

cated image to be quickly set to a default color, such as "paper white".

When an image is allocated using *imem_alloc*, a fill color is specified for the image. All tiles in the allocated image are *effectively* filled with the specified color. This is done by mapping all virtual tiles in the image to the same physical tile. The physical tile is filled with the specified color. Any border tiles around the image are permanently marked as read-only, no-fault on write. Any virtual tiles that overlap the *image_area* rectangle are marked as read-only and copy-on-write (see Figure 9).

Thus, a read of any pixel in the newly allocated image returns the fill color. A write to any pixel in a border tile is ignored, since all such tiles are marked no-fault on write. A write to any pixel in the *image_area* rectangle causes a tile-fault, since all such tiles are marked read-only.

The tile-fault handler detects that the virtual tile is marked copy-on-write, allocates a new physical tile, fills it with the color found in the read-only tile, re-maps the virtual tile to the new physical tile, and resets the protection bits for the virtual tile according to the original value of *prot* from imem_alloc. On return from the tile-fault handler the written-to pixel is updated in the newly mapped physical tile. Subsequent accesses to any pixels in the same virtual tile are directed to the newly mapped physical tile.

Kernel Software

Tile Faults

Tile faults can be generated by a process running on the host processor or by a hardware accelerator on the system bus. Faults generated by a process on the host processor cause a bus error to be received by the host processor, and are handled in a conventional fashion. Faults generated by a hardware accelerator cause a bus error to be received by the hardware accelerator, which generates a vectored interrupt to inform its device driver. These faults are handled by the tile-in daemon, which is discussed below.

Tile Swap I/O

Tile-in and tile-out are handled separately from SUNOS page-in and page-out. Under SUNOS versions 4.0 and 4.1, allocation of swap blocks is delayed until a page-out is required. Swap blocks can therefore appear in random positions on the swap device. In addition, swap blocks are equal in size to the virtual page size, which is 4 or 8 kilobytes.

To obtain maximum tile swap performance, we swap individual tiles to or from contiguous 64 kilobyte portions of the swap device. In addition, the current implementation allocates tile swap space when image memory is allocated, so that the entire image uses a contiguous portion of the swap device[3]. Both of these techniques minimize disk seek time during tile-in and tile-out. Tile i/o is done using a greatly simplified version of *physio* — the block i/o service routine of standard SUNOS. The resulting tile i/o subsystem is very efficient, and achieves approximately 95 percent of the theoretical disk bandwidth during operation.

Tile Daemons

Two new kernel processes are run to support the tiled virtual memory system. The first process, the *tile-out daemon*, tries to maintain an adequate supply of physical tiles in the free list to satisfy future tile faults. This process is similar to the page daemon process, and uses a conventional high/low water mark approach. Tiles that have been scavenged by the tile-out daemon are placed on a FIFO free list, and the virtual to physical mappings are maintained in a *tile cache*. If there is a tile fault on a tile that has recently been scavenged by the tile-out daemon, the physical tile is simply removed from the free list and the virtual to physical mapping restored.

The second process, the *tile-in daemon*, handles tile faults incurred by hardware accelerators. The hardware accelerators perform various image-processing operations on image memory, and are designed to be restartable. If one of the accelerators incurs a tile fault while processing an image, it generates an interrupt and halts. The interrupt causes the appropriate device driver interrupt routine to be invoked.

The interrupt routine gets the virtual tile address of the fault from the accelerator, and passes it to the tile-in daemon along with a notify routine. The interrupt routine then exits, leaving the accelerator halted. The tile-in daemon resolves the tile fault on behalf of the device driver[4], and then calls the notify routine. The notify routine then touches the restart bit on the accelerator, and the accelerator resumes its processing at the previously faulted address. The entire process takes just 200 microseconds on a SUN 4/330.

If this technique was not used, all the virtual tiles needed by an accelerator would have to be mapped and locked to physical tiles before the accelerator was started, similar to the way that pages are locked in prior to a disk transfer. But this is infeasible, given the potential size of the images and the lack of any knowledge of what tiles an accelerator is going to access. It would even be necessary to know which pixels were to be read, and which were

[3]In principle, this approach could cause problems with fragmentation of the image swap space, but this has not proven to be a problem in practice.

[4]The tile fault must be resolved by another process, since a driver can not sleep at interrupt level.

to be written, so that copy-on-write tiles could be properly mapped and copied.

Tile Replacement Algorithm

The choice of replacement algorithms for a tiled VM system is problematical, just as with a paged VM system [2][5][9]. The current implementation of the tile-out daemon uses a modified LRU (Least Recently Used) replacement algorithm. Unfortunately, LRU algorithms can perform poorly in common array processing operations [4][7]. If an array is accessed in a linear fashion in paged or tiled virtual memory, an LRU replacement algorithm acts just like a FIFO algorithm. If the array is larger than physical memory, successive passes through the array will cause the entire array to be copied to and from the swap device.

Despite the potential problems with LRU replacement, we have found it to provide acceptable VM performance in practice. The tile cache (discussed above) has proven to be very helpful in improving system performance.

We have experimented with a RANDOM replacement algorithm, and found it to be much slower than the LRU algorithm, even in cases where the LRU algorithm performs poorly. The dismal performance of the RANDOM replacement algorithm appears to be caused by the complete failure of the tile cache — with random tiles in the free list, the tile cache hit rate falls close to zero.

We are currently investigating the use of LIFO (Last-In, First-Out) replacement. If large data arrays are accessed repeatedly, in a sequential fashion, LIFO replacement may be much more effective than LRU replacement. Initial tests have shown that LIFO replacement reduces tile i/o by 25 to 50 percent in our application.

To illustrate this effect, suppose that we have a 20 megabyte array, and that the operating system has 16 megabytes of physical memory available for paging. Now, suppose that the array has already been accessed sequentially at least once. With LRU replacement the last 16 megabytes of the array remain in memory, and the first 4 megabytes are swapped out. With LIFO replacement the valid and swapped sections are reversed, i.e., the first 16 megabytes of the array are valid, and the last 4 megabytes are swapped out.

When the array is processed a second time, using LRU replacement, the initial portion of the array is invalid, and must be swapped in from disk. But for each page swapped in, another page must be swapped out. As a result, the second sequential pass through the array causes every page in the array to be swapped in once and swapped out once, for a total of 40 megabytes of page i/o.

But when the array is processed a second time, using LIFO replacement, the initial 16 megabytes of the array are still valid. No page faults occur until the last 4 megabytes of the array are accessed. Just as with LRU replacement, for each of these pages swapped in, another page must be swapped out. So with LIFO replacement, a total of 8 megabytes of page i/o are required.

In general, for repeated sequential access to an array of N bytes on an operating system with P bytes of physical memory, with $N>P$, LRU replacement will require $2N$ bytes of page i/o, and LIFO replacement will require $2(N-P)$ bytes of page i/o.

We expect to continue investigating replacement algorithms in the future, including LIFO and the most-distant algorithm discussed in Future Work.

VM Performance

By making a trivial change to the kernel code for *imem_alloc*[5], we are able to turn our tiled VM into a conventional paged VM. The tiled version uses 64 kilobyte tiles, the paged version uses 64 kilobyte pages. Every other aspect of the two kernels is identical, including use of the IMMU hardware, fault handling, swap space allocation, swap i/o transfer size, replacement algorithm, and tile/page daemons.

These otherwise identical VM systems are ideal for comparing the performance of tiling and paging on a common image processing operation such as filtering[6]. To do the comparison, we use one of our hardware accelerators to filter a 12 megabyte (4000x3000 pixel) greyscale image. The filtering operation requires a horizontal and a vertical pass, so the entire image is read and written twice: once by rows; once by columns.

Our test filters the image once for each of a range of physical memory sizes, recording the execution time for each. We repeat the test using both the tiled and the paged VM system. At the start of the test, 16 megabytes of physical memory are available to the VM system for image storage. On each pass of the test the following steps are performed:

1. The required images are allocated, initialized, and forced out of physical memory;
2. The filtering operation is performed and the processing time is recorded;
3. The images are deallocated; and

[5]We simply set XSIZE (the number of bits in the X address portion of the virtual address) to 8. This effectively transforms a request for an *N by M* tile image into a request for an *(N x M) by* 1 tile image. That is, the allocated image is 1 tile wide, regardless of the requested width.

[6]Filtering is used for a variety of image processing operations, including rotation, scaling, sharpening, blurring, and warping.

4. The amount of physical memory is reduced by 256 kilobytes.

The passes continue until the VM system starts thrashing.

Figure 10 shows the results of these tests as a graph of execution time vs. physical memory size. The paged VM system starts thrashing when physical memory drops to 13 megabytes[7]. The tiled VM system, on the other hand, exhibits a linear degradation in performance until physical memory drops to 2 megabytes. Below 2 megabytes there is a steeper but still roughly linear degradation.

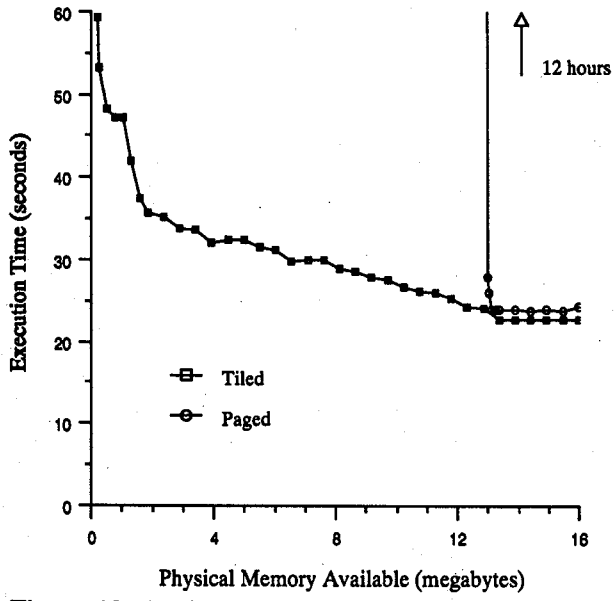


Figure 10: Performance of VM System on Filtering Task

For this filtering task, the paged VM system requires at least 13 megabytes of physical memory. The tiled VM system continues to function with 1/4 megabyte of memory and only a doubling in the nominal filtering time. This is a 50-fold reduction in working set size.

Future Work

Tile Replacement Algorithms

As discussed in the *Kernel Software* section, the tile-out daemon uses a modified LRU algorithm, and this algorithm can perform poorly for some common array access patterns.

Wada [9] suggests the use of a *most-distant* replacement algorithm for image processing applications. In this algorithm, the tiles most distant from the last accessed tile are candidates for replacement. This algorithm appears to be well behaved under many of the common access patterns in image processing. However, it is more computationally expensive than other common replacement algorithms, and also requires knowledge of the last tile accessed. This information is not available from our IMMU hardware.

We would like to investigate whether an approximation to the most-distant algorithm would be effective. One possible approach would be to use a conventional clock algorithm to clear all of the tile reference bits at regular intervals. When the tile-out daemon runs, the centroid of the recently referenced tiles could be used as an approximation to the most recently used tile.

Tile Prefetch Algorithms

The current tiled VM system is *demand tiled* — a tile is fetched from the swap device only when a tile fault occurs. It may be desirable to modify the tile fault handler to detect common image access patterns (such as row or column access), and then asynchronously prefetch entire rows or columns of tiles before they are needed.

Alternatively, it may be easier to add a system call that would allow the application process to provide hints to the kernel about intended image access. The hints should include the region of the image to be processed and the image access pattern.

We would also like to investigate the possibility of coupling images with respect to tile faults, so that a tile fault on one color component of a picture would cause an implied fault on the corresponding tiles in the other color components. Similar coupling of source and destination images may also be useful.

Extensibility to Higher Dimensions

The current implementation of the IMMU hardware and software provides an efficient environment for processing large 2-dimensional arrays. However, the array virtual to tiled virtual address translation (as performed by the IMMU) is easily extensible to higher dimensions. The only fundamental constraint is the size of a virtual address on the host processor.

Most modern processors provide 32-bit virtual addresses. In the current implementation, both the X and Y portions of an array virtual address can be up to 16 bits. A tiling architecture to handle 3-D or 4-D arrays would require reductions in the maximum dimensions of the arrays, but would still be useful for some applications. However, future trends are definitely toward larger virtual addresses [8], and 64-bit machines are already becoming available. On such a machine, a tiling architecture for 3-D or 4-D arrays would be quite interesting.

Integrated MMU and IMMU

The IMMU is completely separate from the host processor's MMU and is situated on the system bus. This limits the potential performance of the image memory system, due to the overhead in

[7]This is the size of the image plus free space for the VM system to use for tile fault resolution.

addressing the system bus and the inability to take advantage of the cache hierarchy on the host processor.

The IMMU is very similar to a conventional MMU, with the exception of the array virtual address to tiled virtual address mapping. Integration of this mapping logic into a conventional MMU should be straightforward, but is beyond our means.

Conclusions

We have implemented a tiled virtual memory for UNIX, based on a custom MMU and supporting kernel software. Together, they provide a tiled, shareable, virtual memory that has proven to be an efficient environment for manipulating 2-dimensional arrays of data.

Use of the tiled VM presented here is nearly transparent to application software and programmers. In fact, the same executable binaries run on systems with and without tiled VM, the only difference showing up in array handling performance. For arrays that fit in memory, tiled and paged VM are equal in performance. For arrays larger than physical memory, tiled VM provides much higher performance.

This tiled virtual memory has been used successfully in a commercial product for the color electronic prepress industry. Although we have used this tiled VM solely for image processing applications, we believe it would be equally useful in a variety of other applications.

Acknowledgements

Gary Newman was responsible for the conceptual design of the IMMU and its use in a tiled virtual memory system, and later guided the development of both the IMMU and the kernel software described here. Steve McLafferty and Bob Getz implemented the IMMU hardware.

References

[1] Blinn, J. F., "The Truth About Texture Mapping", IEEE Computer Graphics and Applications (March 1990), 78-83.

[2] Denning, P. J., "The Working Set Model for Program Behavior", Commun. ACM 11(5) (May 1968), 323-333.

[3] Denning, P. J., "Virtual Memory", ACM Comput. Surv., 2(3) (Sept 1970), 153-189.

[4] Hatfield, D. J., and Gerald, J., "Program Restructuring for Virtual Memory", IBM Systems Journal 10(3) (March 1971), 168-192.

[5] Madnick, S. E., and Donovan, J. J., *Operating Systems*, McGraw-Hill, New York, 1974.

[6] McKellar, A. C., and Coffman, E. G., "Organizing Matrices and Matrix Operations for Paged Memory Systems", Commun. ACM 12(3) (March 1969), 153-165.

[7] Ryman, A., "Personal Systems Image Application Architecture: Lessons Learned from the ImagEdit Program", IBM Systems Journal 29(3) (Sept 1990), 408-420.

[8] Slater, M., "Is 64 bits the next step?", Microprocessor Report, 5(4) (March 1991), 3-4.

[9] Wada, B. T., "A Virtual Memory System for Picture Processing", Commun. ACM 27(5) (May 1984), 444-454.

Author Information

Jim Franklin received a BA in Mathematics from Cornell University in 1974, and an MS in Computer Science from the University of Maryland, College Park in 1976. He was a member of technical staff at Bell Laboratories until 1980, working on software tools, and then joined Automatix, working on programming languages for robotic systems. In 1986 he joined Kodak Electronic Printing Systems, where he works as a consulting engineer in the Color Image Products Group, investigating operating system issues for new imaging platforms. He can be reached via U.S. Mail at Kodak Electronic Printing Systems; 164 Lexington Road; Billerica, MA 01821. His electronic mail address is jwf@keps.kodak.com or uunet!keps.kodak.com!jwf .