RC 20616 (90921) 31 July 1996 **ComputerScience**

IBM Research Report

A Study of Virtual Memory MTU Reassembly within the PowerPC Architecture

Lucas Womack
University of Maryland
College Park, MD

Ronald Mraz
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Abraham Mendelson
Technion Institute of Technology
Israel

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

A Study of Virtual Memory MTU Reassembly within the PowerPC Architecture

Lucas Womack – University of Maryland, College Park, MD
Ronald Mraz – IBM T.J. Watson Research Center, Yorktown Heights, NY
Abraham Mendelson – Technion Institute of Technology, Israel

Abstract

Message transfer unit (MTU) reassembly schemes in modern operating systems cause I/O performance degradation when MTU sizes are larger than the architecture's page size. This can happen with emerging network technologies, such as Asynchronous Transfer Mode (ATM), where MTUs can be 64 KB or greater. Traditional solutions either reassemble using memory copy or preallocate contiguous memory; these, however, lack speed or consume excess resources, respectively. This paper presents an alternate scheme called Virtual Memory MTU Reassembly (VMMR) which reassembles non-contiguous pages through virtual memory remapping. VMMR allows hardware/software interfaces to efficiently DMA large MTUs in hardware pages and remap them to a contiguous address space. Studies done on a PowerPC 601 show that this method can outperform memcopy by one to two orders of magnitude (the maximum VMMR bandwidth is 14.7 Gbits/sec). High-performance multimedia applications, such as video on demand and video conferencing, can greatly benefit from such a performance boost.

1. Introduction

Emerging network technologies, such as Asynchronous Transfer Mode (ATM), have been the focus of manufacturing efforts aimed at producing fully integrated, high-performance solutions **[1]**. Chipsets like the IDT77201 NICStAR™ **[6]**, for example, support segmentation and reassembly of ATM cells, hardware checksumming, and DMA protocols that specify sequential cell placement in predefined, possibly discontiguous, hardware pages. The latter feature eliminates the need for an operating system (OS) to copy data between host memory and a network card during data transmissions, yielding what is known as a zero-copy interface.

For a small message transfer unit (MTU – also known as a maximum transfer unit or a hardware layer protocol data unit – PDU), this DMA approach offers substantial performance gains. Many modern networks, however, have been defined to support large MTU sizes. The ATM specification, for example, allows MTUs up to 64 KB in length **[1]**. FDDI calls for MTUs of 4,352 bytes, and IP over ATM requires 9,180 bytes **[13]**. When large MTUs are streamed into a receiver's memory on an architecture with small page sizes, overall I/O performance can actually drop. This is because the OS needs to concatenate and reassemble pages belonging to an MTU before it can send it up to the protocol stack for additional processing. Systems that implement reassembly using copy semantics suffer the most since they require an additional slow memory copy (memcopy) at the device driver level.

General issues involving memory-to-memory copies **[5]** and network-to-memory copies **[3]** have been known to represent a significant portion of the overhead associated with network data transport services. Much of the literature, however, has tended to concentrate on maximizing buffer reassembly performance at the protocol layers, where messages (MTUs) are combined into an application data unit, or ADU. MTU reassembly has, on the other hand, received far less attention. A widely adopted ad-hoc solution is to copy each fragmented MTU into a separate, pre-allocated, contiguous, MTU-sized buffer. This method is not only wasteful in terms of memory usage and CPU time, but also slow. Using a similar strategy, Traw and Smith studied data movement from a fast ATM card they designed to system and user spaces in the AURORA Testbed environment **[2]**. Their resource inefficient technique required a kernel pre-allocation of two 64 KB contiguous pinned buffers. One of the inevitable findings of that research was that overall bandwidth was highly dependent on data copy performance.

Virtual memory remapping has become a logical solution to the copy problem. It is an attractive scheme because the system "moves" data by altering page table entries rather than performing physical copies **[4]**. Tzou and Anderson measured the impact of remapping virtual addresses in message passing environments **[11]**. Their study showed that the use of buffers and preallocated virtual and physical address regions limited VM remapping performance gains over data copy. As a result, the authors suggested limiting generalized use of the virtual address space by mapping communications buffers to a fixed virtual address range shared by all processes. This solution is not general enough, however, for use in fast, next generation communications systems. Other papers, such as **[12],** provide similar performance observations but only offer additional non-generalized buffer-based solutions.

Druschel and Peterson recently proposed another such technique known as *fbufs*, or fast buffers, to mitigate VM remapping expenses **[12]**. Fbufs are allocated by the operating system from a shared memory pool and sent to device drivers or applications for incoming and outgoing data. The buffers are allocated and reused in such a way so as to minimize TLB and cache flushes, memory management unit (MMU) updates, and protection domain traversal overheads. An implementation of fbufs in Solaris by Thadani and Khalidi showed that network throughput improved by more than 40% and CPU utilization dropped by more than 20% **[10]**.

The problem with the fbuf scheme is that it assumes the OS communicates with "deep" adapter interface cards (i.e. entire MTUs are buffered in card memory for subsequent transfers). When a complete MTU has arrived and been reassembled, a deep interface card notifies the device driver on the receiver, which then analyzes header information stored on the card so as to fetch an appropriately sized buffer from the kernel. In the fbuf scheme, data residing on the network card is then streamed into the contiguous buffer and then copied to a pageable fbuf accessible to kernel and user processes. In next generation ATM hardware, however, MTUs are pipelined across the network to achieve much faster throughputs and lower latencies – the result is a thin interface (i.e. memory efficient). One implication is that MTUs are DMAed into physically discontiguous pages and require reassembly prior to protocol stack processing. This can be achieved either through hardware support, special kernel routines, or a combination of both.

One such hardware assist is known as DVMA, or Virtual Direct Memory Access, and it was devised at Sun Microsystems to speed up data flow between I/O devices and memory **[13]**. An operating system can program an I/O memory management unit (IOMMU) on the DVMA controller to set up appropriate virtual to real translations so that an incoming I/O stream is automatically written to contiguous virtual space (discontiguous physical space). The problems with this approach are twofold. First, the IOMMU has a fixed number of address translation lines and can only reassemble a message of limited size. The STP2220BGA UltraSPARC to SBus interface chip, for example, has a 16 entry IOMMU that can support up to one megabyte of remapped data **[14]**. Second, DVMA requires consistency between its page tables and those on the host OS; there is significant overhead in setting up these translations and in maintaining coherency.

In this paper, we present a *software* solution known as Virtual Memory MTU Reassembly (VMMR) and show it to have a maximum bandwidth fifteen times higher than the STP2220BGA hardware **[14]**. We claim that when large MTUs span multiple discontiguous pages in main memory, data can be efficiently reassembled on the receiver's side of a thin adapter card using VMMR before the MTU is sent up to higher protocol layers. VMMR relies on the hardware VM subsystem to remap virtual memory page pointers. The hardware page table then provides the transformation between contiguous virtual addresses and discontiguous physical addresses.

The PowerPC (PPC) architecture provides a unique virtual memory subsystem that can be used efficiently to support VMMR. The 52-bit PPC virtual address space spans 2^{24} 256 MB segments, each denoted by a unique software-controlled ID (VSID) **[8]**. The correlation between segments and virtual addresses provides a fast, natural mechanism for sharing information among processes. We suggest that, instead of creating and maintaining communication buffers, a segment addressable by all process address spaces be allocated as a shared pool of virtual pages used to move information among process domains. Note that VMMR does not require pre-allocated buffers, as was purported to be the major reason for performance loss in other zero-copy techniques.

A VMMR scheme using a virtual page bitmap pool has been implemented to collect quantitative results, including a VMMR lower bound, on a PowerPC 601. It has proven to outperform traditional network-to-memory or memory-to-memory data copy by a factor of nearly 83 (14.7 Gbits/sec) in best case benchmarks. Furthermore, VMMR can reassemble very large data sets without additional device setup and coherency overheads, unlike hardware solutions such as DVMA.

In the next section, we formalize the measurement techniques, describe the execution environment, and outline the experimental method used in this paper. Section 3 discusses demand paging in the PowerPC 601 architecture. Section 4 presents results obtained from the benchmark code used to quantify VMMR performance, and finally, Section 5 summarizes our findings and conclusions.

2. Experimental Framework and Environment

This section presents an outline of the research methodologies used in this study's performance evaluations. To avoid limitations due to specific implementations of any one operating system, a simple yet powerful kernel called NAP was used to benchmark VMMR with no OS overhead. Operating system implementation considerations are discussed in section 4.4.

2.1 NAP

A *naked application* is a program that runs on hardware without the support of an intervening operating system. NAP is a Naked APplication support environment created at IBM to facilitate PowerPC-based system software development. It has been used in the optimization of network interfaces, in low latency commodity networks, and in guaranteeing quality of service at the microsecond level.

NAP runs native to the PowerPC (PPC) 601, and its serial port output interface was used to send experiment results to a file on a receiving PC. Although NAP provides elementary interrupt hooks and exception handling, the code used in this research relied solely on NAP's bootstrap, serial output, and timer interrupt handler routines. Since there was no underlying OS support, all experiments ran in supervisor state as standalone processes. This allowed time sensitive code (e.g. benchmarks) to collect data unaffected by context switch overhead. The only variance any code running atop NAP could experience would be due to the nonmaskable PowerPC timer, which interrupts 18 times per second. NAP services each interrupt and returns to the executing process within 3 ms. This was tolerable since all experimental data collected in this research were cumulative timing averages, as described in Section 2.5.

2.2 Experimental Method

Figure 1 presents a diagram of the test machine memory layout that was selected for the execution environment. Memory was partitioned into distinct caching regions belonging to *supervisor space* or to *page table space*. This was done to analyze VMMR and memory copy (memcopy) performance under varying caching conditions. All addresses within these regions were initially mapped in virtual memory in a one-to-one virtual-to-real manner.

The first four megabytes of memory, which contained benchmark code and NAP support routines, were configured as *supervisor space*, with the exception of a 512 KB page table placed at the 2 MB boundary. During benchmark initialization, an 8 KB free page pool bitmap belonging to *page table space* was written to

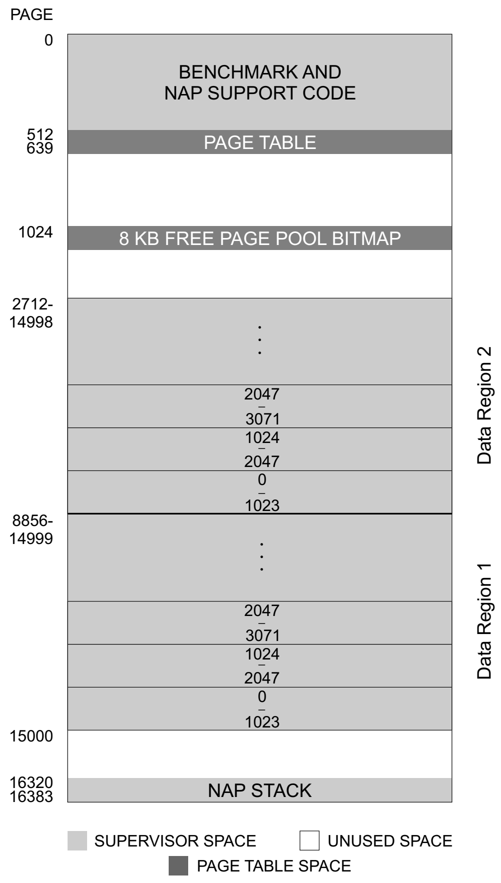

Fig. 1. Test Machine Memory Utilization

memory at the 4 MB boundary (this is discussed in detail below). A 64 KB *supervisor space* region was then reserved at the bottom of memory for NAP's stack.

Data pages used in the benchmarks were included in *supervisor space* and placed in a memory region that varied in size from test to test. Data region 1, shown in Figure 1, consisted of pages filled with word counts as illustrated. The purpose of all tests, with the exception of the *memcopy* benchmark, was to reorder these pages in virtual memory to create a virtually contiguous region of increasing word counts (i.e. to simulate an MTU reassembly). Alternately, the *memcopy* test timed a fast *word* copy given source and destination memory addresses; during this benchmark, data region 1 was replicated in data region 2. With approximately 60 MB of memory left for benchmark data storage (64 MB - 4 MB of code), the data regions' sizes were limited to 24 MB (6,144 pages) each.

VMMR performance evaluation consisted of data analysis from four test categories. The *total caching* tests measured VMMR and memory copy times for L1 cached *supervisor* and *page table* spaces. *Kernel caching* and *page table caching* tests, on the other hand, collected times for L1 cached *supervisor* and *page table* spaces, respectively. The *no caching* tests ran without the assistance of L1. Each category, with the exception of *total caching*, was broken down into four distinct tests as follows. *Memcopy* measured the times the hardware required to copy a set of contiguous pages into another contiguous block, given source and destination pointers. *No search* benchmarked calls to the VMMR reassembly code when a search for a free contiguous page set from a pool of virtual pages (discussed in the next section) was not required. *Minimal search* and *worst case search* tests yielded MTU reassembly times for two page pool search extremes. In addition to the above, a *random search* test was developed for the *total caching* category that gave performance feedback for a simulated environment where page pool fragmentation was modeled over time.

2.3 Virtual Page Pool Bitmap

As discussed in the introduction, the VMMR technique rearranges discontiguous physical pages by manipulating virtual page pointers in the hardware page table. Although physical pages remain discontiguous in main memory, they are made contiguous in virtual memory, and the page table provides any necessary virtual-to-real address translation. For the technique to work, an operating system has to maintain some pool of free contiguous virtual pages. We quantified VMMR performance by using an 8 KB free page pool bitmap to represent all pages in a reserved 256 MB segment (i.e. each bit corresponds to one 4 KB page). Available pages in the segment are found by a search routine which looks for a set of contiguous free bits of length *m* within the bitmap, where *m* is the MTU size in pages. Successful searches yield a bit number representing some starting page within the segment; unsuccessful searches return an error code to the caller.

Contiguity is defined as some continuous sequence of unallocated bits within a word or across several words. The top half of Figure 2 illustrates contiguity within a bitmap word; the bottom half, across multiple words. With the exception of the *random search* test, all searches begin at the most significant bit of word 0 and proceed sequentially to its least significant bit and then to the most significant bit of the following word in memory. In the *random search* benchmarks, searches begin at some random word within the bitmap to minimize biasing effects due to fixed starting locations. In the event that the search pointer has prematurely arrived at the end of the bitmap without having checked all words in the bitmap or having found a suitable free page set, it is reset to word 0, and the search continues as described above.

The bitmap search routine was written in assembly for improved efficiency and uses a few instructions peculiar to the PowerPC architecture to speed up the bitmap searches. The algorithm consists of an outer loop that retrieves successive words in the bitmap and that locates possible free page set candidates, and it also contains an inner loop that examines those candidates for contiguity as de-

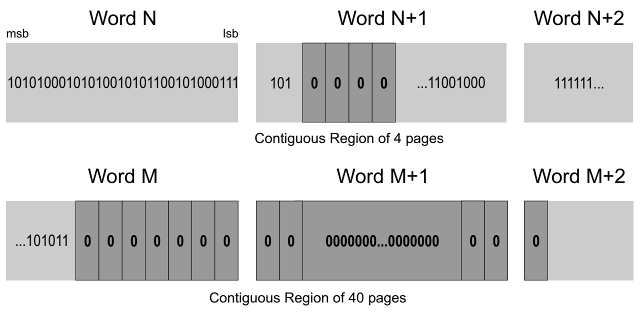

Fig. 2. Contiguity Definition for Free Page Pool Bitmap

fined above. If the bitmap is highly fragmented, the search algorithm loses performance because it stays in the inner loop for long periods trying to resolve all bit transitions it encounters.

2.4 Experimental Environment

Benchmark and VMMR code compilation were done on an IBM POWERstation 530 running AIX 3.2.5 using the IBM Xlc 1.3 compiler; flags for standard optimization (-O) and for PowerPC 601 code generation (-DPPC601) were specified. The base test machine was a PowerPC reference platform (601) **[7]** with 64 MB of main memory and no L2 cache. All benchmarks ran with address translation enabled; we used the recommended minimum page table size of 512 KB for a 64 MB machine. Experimental data (timings) were sent through the serial port to a PC and captured in files using Procomm Plus for DOS. The results were then filtered and analyzed using custom programs that employed the measurement techniques described below.

2.5 Performance Metrics

All tests, with the exception of *random search*, collected reassembly times for about eighty MTU sizes m, ranging from 1 page (4 KB) to 6,144 pages (24 MB). Thirty-two iterations were performed per MTU size, with L1 and the TLBs flushed on each iteration, and their arithmetic mean reported as the final time.

Each *random search* test collected reassembly times for a given set of 3,000 randomly allocated page pool bitmaps; 32 iterations (again with L1 and TLBs flushed on each iteration) were run per bitmap and their times averaged to yield a final reassembly time for that bitmap. Output files for the *random search* tests were also filtered through a program which extracted absolute maximum and minimum times across all 96,000 iterations and which averaged final reassembly times for bitmaps that were 40% and 10% allocated (see section 4.3). Because of the time required to run each *random search* test (2 to 16 hours), only about 20 MTU sizes were selected from the 1 to 6,144 page range.

The free page pool search algorithm uses a bitmap representation of a 256 MB segment of virtual pages reserved for benchmark data reassembly. Since each bitmap bit represents one 4 KB PowerPC 601 page, there are 2^{16} bits in the bitmap. We define the bitmap allocation level as follows:

$$l = \frac{\text{No. of allocated bits in bitmap}}{65{,}536}.$$

For example, if an application uses this page pool bitmap for memory allocation, $l = .75$ would mean that 75% of a PPC 601 segment is currently in use and not available to other processes.

Because NAP does not provide a floating-point library, we opted to use a simple Fibonacci sequence random number generator that would work well within the limits of 32-bit integer arithmetic **[9]**:

$$X_{n+1} = (X_n + X_{n-1}) \bmod 7\text{FFFFFFF}_{16}$$

where 7FFFFFFF_{16} is the largest representable 32-bit integer and X_0 and X_{-1} were arbitrarily set to 17 and 37, respectively. Since this sequence produces an even distribution of integers, the resulting random sample is sufficient for the scope of this research.

3. PowerPC 601 Demand Paging

The PPC 601 uses a sophisticated address translation mechanism combining paging and segmentation. In this architecture, 32-bit logical addresses are transposed to a 52-bit virtual address space and subsequently converted to 32-bit physical addresses. Sixteen 256 MB segments divide the 32-bit logical address space, and each segment is partitioned into 2^{16} 4 KB pages. To speed up address translation, the memory management unit (MMU) includes two translation lookaside buffers. The first, a 256 entry two-way set associative TLB, is unified (UTLB) and contains virtual to physical translations for instructions and data. The second is a four entry, fully associative, instruction TLB (ITLB).

When presented with a 32-bit logical address, the MMU uses the four most significant bits LA0-LA3 (big-endian notation) to index a 16 entry segment register table (see Figure 3) **[8]**. Each entry stores, among other information[†], a 24-bit virtual segment ID (VSID). This number, along with the next 16 logical address bits LA4-LA19, form a 40-bit virtual page number (VPN), which is then paired with the remaining 12-bit logical offset to yield a 52-bit virtual address. The ITLB, UTLB, or the operating system supplied page table contains the translation from VPN to PPN, the 20-bit physical page number.

The ITLB (instructions only) and UTLB (instructions and data) are indexed using LA4-LA19; if either return a valid PPN, the hardware translation process ends with a 32-bit physical address consisting of the PPN and the 12-bit offset. If both fail to yield a PPN, the translation process continues with a page table search.

The PPC 601 page table is written to and updated in memory by the operating system. Its location in memory and its size are known to the hardware via the Table Search Description Register (SDR1), which is also maintained by the resident OS. Bits 0-15 of SDR1 specify the base address of the table, which consists of a number of page table entry (PTE) groups of eight 64-bit PTEs each. One PTE defines a VPN to PPN translation and includes bits that the OS can use to facilitate its replacement policy.

Locating a PTE within the page table requires the generation of up to two hashing functions **[8]**. Bits 5-23 of the VPN (the VSID) are exclusive-ORed with bits 24-39 (the 16-bit logical page index) to create the primary hash. When needed, the secondary hash can be calculated by taking the one's complement of the primary hash. The 19-bit hash loosely represents an index into the PTE groups of the page table. The previous statement is qualified because the PPC 601 page table can vary in size from 64 KB (2^{10} PTE groups) to 32 MB (2^{19} PTE groups), a variation of 2^9. Bits 0-8 of the hash are therefore ANDed with a hash table mask stored in bits 23-31 of SDR1. This value is then ORed with the least significant 9 bits of the page table base address to yield a value that is used to generate a physical address for a PTE group.

Once the PTE group address is calculated, the hardware begins a sequential search through the group for a valid PTE. The MMU compares the current VSID and abbreviated page index (the most significant 6 bits of the logical page index) with the VSID and API stored in the PTE (see the PTE blowup in Figure 3) **[8]**. If they

[†] This section presents a *general* overview of the address translation mechanism in the PowerPC 601 and assumes that block and I/O translations do not apply. See **[8]** for more information.

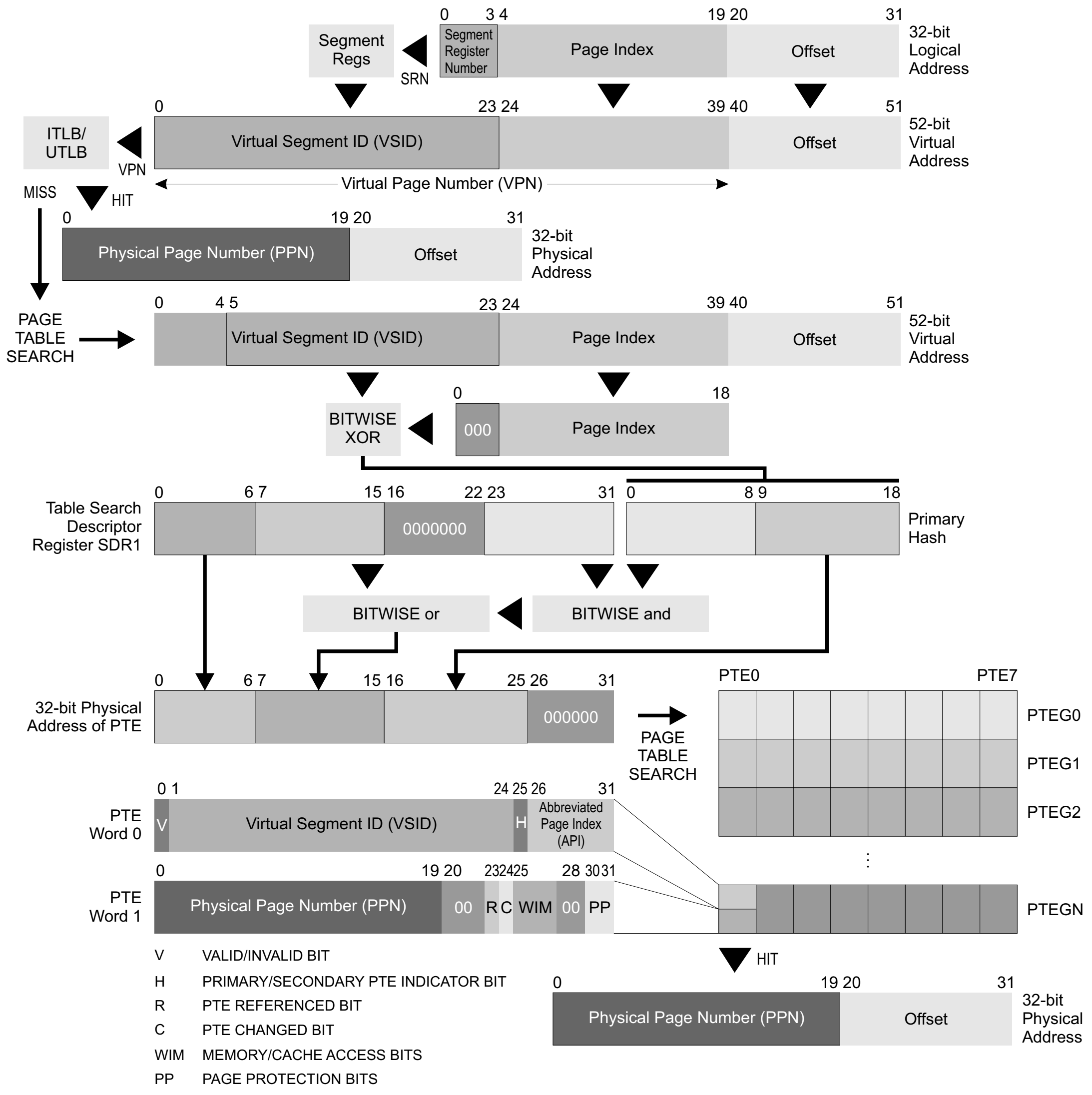

Fig. 3. PowerPC 601 Address Translation [8]

match, and if the PTE is marked valid and primary, the PPN is extracted and concatenated with the 12-bit offset to yield the final translated physical address. If the PTE group does not contain a matching PTE, the hardware calculates the secondary hash as described above, repeats the masking process, and searches through a secondary PTE group for a matching secondary PTE. In case a valid translation cannot be found, the hardware faults with a page table exception and, once the OS has updated the page table, repeats the entire translation procedure.

The VMMR code allocates page table entries using the algorithm outlined above. In addition, benchmark caching policies are implemented efficiently using the "I" bit in the memory/cache access control bits (these are marked as "WIM" at the bottom of Figure 3). The "I" bit, also known as the cache invalidate bit, controls L1 (and L2) lookup for the PTE's associated physical page. All PTEs in the no caching tests and PTEs for uncached memory regions in the kernel and page table tests had this bit set.

4. PowerPC MTU Reassembly

To determine VMMR performance benefits in the PPC 601, a benchmark suite was run for varying caching and page pool fragmentation configurations. This section elaborates on those tests and analyzes their results.

4.1 Memory Copy vs. VMMR

The first goal of the research was to generate data comparing the performance range of a fast memcopy with that of a sample VMMR approach. We decided to bias algorithm choices in favor of memcopy, which is often used for message reassembly in most operating systems.

The memcopy benchmark consisted of a routine that copied anywhere from 1 to 6,144 contiguous pages (word by word) to an alternate contiguous location in memory. Address translation was

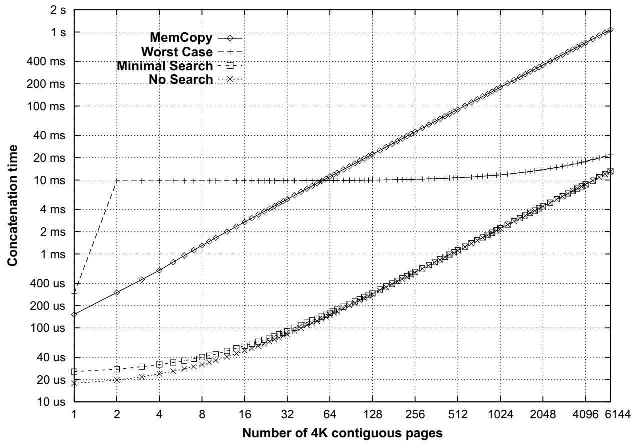

Fig. 4. ***Total Caching* Benchmarks**

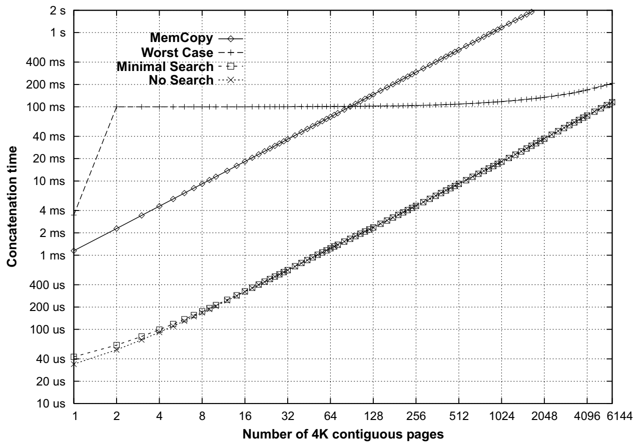

Fig. 5. ***No Caching* Benchmarks**

enabled during this test, with PTEs set so that all source and destination block addresses were mapped in a one-to-one virtual to real correspondence. Although the performance of this scheme is hard to achieve in a memcopy implementation under UNIX, it yielded data for a best case scenario that was used in developing objective performance comparisons.

The design of the VMMR code was based on two factors. First, we wanted to develop routines, that, while not unreasonably slow, were inefficient enough to demonstrate how a second-rate algorithm could fare against a fast memcopy. Second, and perhaps most importantly, we wanted to write simple, easy-to-debug code. As mentioned in Section 2.3, the final choice was a virtual page pool bitmap, with each bit representing one page of a reserved 256 MB segment. It should be pointed out at this time that a VMMR scheme based on a free page pool bitmap is NOT an optimal one. Other organizations, such as the buddy system and AVL trees **[15]**, are faster and more efficient. The goal of this research is to collect quantitative data showing how well VMMR *could* fare against traditional solutions; the data presented here should be used as a guide in developing appropriate VMMR techniques suitable for some OS and architecture combination.

In the *minimal* and *no search* tests, all bits in the free page pool map were freed (0% allocated). The former test measured the time required by the search routine to find a free block of contiguous bits at the start of the bitmap, plus the time needed to allocate appropriate PTEs for VMMR. *No search*, on the other hand, yielded data for a VMMR best case. It only measured the time required by the search routine to allocate the same PTEs. Thus, *no search* also represents an absolute lower bound for *any* VMMR implementation (up to the efficiency of the allocation algorithm).

The *worst case* benchmark timed the bitmap search and PTE allocation code for a bitmap with maximum fragmentation. For an MTU size of one page, the entire bitmap was allocated with the exception of a single bit at the end. For other MTU sizes, the bitmap was written with alternating ones and zeros; a set of contiguous free bits matching the MTU size was then placed at the end. The *worst case* test stressed the VM reassembly algorithms under highly improbable conditions; the *random test*, discussed in the next section, was devised to address this.

Figure 4 presents the data collected from four tests in the *total caching* category. Note, upon inspection of the *memcopy* and *minimal search* curves, that there is a transition from a fivefold to near eighty-fold increase in performance at the 1 and 6,144 MTU page sizes, respectively. Speedup increases with larger MTU size because the initial call to the reassembly subroutine is amortized over many pages. The *minimal* and *no search* tests behave as expected; the convergence of the two data sets at the 32 page MTU size boundary demonstrates how the PTE allocation time quickly dominates over the time for a minimal search of available contiguous bits. Note that at 6,144 pages, the maximum VMMR bandwidth for *minimal search* is 14.7 Gbits/sec. *Memcopy* can only claim 176.9 Mbits/sec.

The *worst case* test lags behind *memcopy* for MTU sizes between 2 and approximately 60 because the search algorithm may need to resolve close to 2^{16} bitmap transitions. Even at the one page MTU size, *worst case* results are slower than *memcopy* (289 ms vs. 143 ms). This, however, is to be expected since *memcopy* copies 1,024 words while *worst case* has to evaluate 2,047 fully allocated words, one partially allocated word, and a pre-allocated free bit at the end of the bitmap. As MTU size increases (2 pages and above), the *worst case* curve follows a flat plateau (where the search time, which is roughly constant, dominates) and then rises gradually to follow the *minimal* and *no search* curves. In this region, the time to allocate PTEs in the page table begins to dominate despite a reduction in search time.

Figure 5 shows the results of the *no caching* test category. The only noticeable difference between this group and the *total caching* test group is that times have uniformly risen due to the lack of L1 caching. The *minimal* and *no search* curves stabilize to a final slope much faster than those in the *total caching* category also because of no L1 support. Initial and final speedups are approximately 33 and 64, respectively, as compared to 5 and 83 in the previous case. The initial increase in speedup is attributed to the slow performance of L1 unassisted memcopy. As the search algorithm and reassembly code spend much time in longer loops, VMMR loses some performance and subsequently settles to a constant speedup over *memcopy* for MTU sizes of 16 pages and above. One other observation appears to support these conclusions – the *worst case* curve, which is the result of a code-intensive search, has a twelve-fold performance loss from *total caching* to *no caching*; *memcopy* only experiences about 60% of that slowdown.

Results of the *kernel caching* and the *page table caching* tests were not shown because they were indistinguishable from those of the *total caching* and the *no caching* tests, respectively. This is interesting because it points out that page table and bitmap accesses are not dominant factors in the benchmark timings. The one-to-one correspondence between *kernel* and *total caching* is to be expected; as seen in the previous paragraph, code caching is critical to maximizing VMMR performance.

4.2 The Random Test

Since the *worst case* and *no search* tests represent theoretical extremes, a more refined test was needed to get a realistic view of the performance advantage in the VMMR approach. In the second part of this research, a benchmark was designed to simulate evolving fragmentation levels within the page pool bitmap. Twenty-three MTU size tests were run, and each test consisted of measuring VMMR times for 3,000 randomly allocated bitmaps. Thirty-two timing iterations were run per bitmap, with their average reported as the final time, for a total of 96,000 iterations per test. Each iteration, in turn, measured the time required by the search routine to locate a set of contiguous bits from the most significant bit of one of 2,048 random words within the bitmap (plus the time for appropriate PTE allocation). As with the *worst case* benchmark, the search algorithm was always guaranteed to find a pre-unallocated page set at the end of the bitmap.

In each MTU size test, the first thousand (of 3,000) page pool configurations (i.e. randomly allocated maps) were obtained by calling the random number generator described in section 2.5 500 times per map. All bits in the map were initially marked as allocated, with the exception of a pre-unallocated page set at the end. Each call to the random number generator returned a random bit position within the bitmap which was then marked free. Figure 6 plots the number of allocated bits for each of the 3,000 page pool configurations in the 16 page MTU size test; the leftmost third shows the deallocation sequence for the first thousand configurations. Note that the process outlined above yielded maps that covered the entire allocation range; however, a fairly large number of tests benchmarked allocation levels less than 50%. To compensate in the next thousand page pool configurations, all bits in the bitmap were marked allocated (with the exception of a pre-unallocated page set at the end), and the random number generator was invoked 50 times per map. This provided a more dense coverage of highly allocated bitmaps, as seen in the middle third of Figure 6; the rightmost third shows additional dense coverage obtained in the last thousand configurations by calling the number generator 25 times per map. The overall effect of this deallocation scheme was that the *random tests* returned more data points for high allocation levels within the bitmap. This was necessary to construct a worst case curve that could be used to gauge performance of the *random* tests against the *worst case* test.

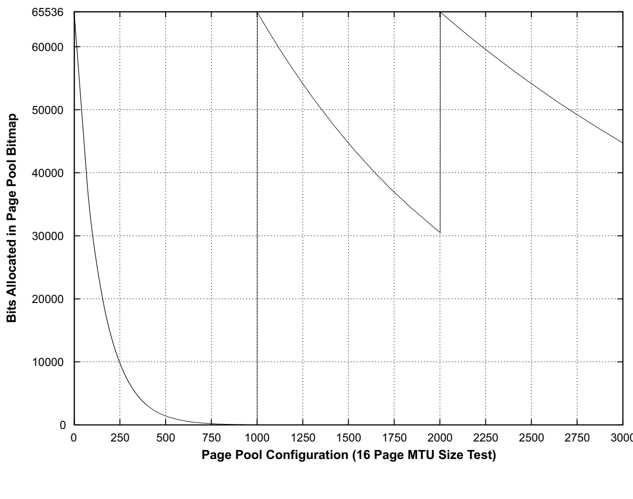

Fig. 6. Page Pool Configurations

Figure 7 shows the results of 12 of the 23 MTU size tests that were run for the *total caching* case. Each plot contains 3,000 data points, and each point is the arithmetic mean of 32 timings. Note that the absolute minimum data point for some MTU size *m* agrees well with *m*'s *minimal search* data point on Figure 4. Also note that the 50% allocation mark falls close to the 32,768 transition line (about half the number of possible transitions); this is as expected when a random number generator is creating satisfactory allocation maps. Allocation levels greater than 50% see data points returning to smaller numbers of binary transitions; as allocation increases, small free areas are coalesced into larger allocated ones, thus reducing the total number of transitions.

For an MTU size of one page, the plot remains flat for all tested allocation levels. This is because the search algorithm has no problem locating a free bit. As the MTU length increases from 2 to 4 pages, allocation levels from 50% to 95% give the search algorithm trouble in locating an appropriately sized free area. This trend continues at 6 pages; here, however, the onset of a behavior present in all remaining plots is visible. For allocation levels of 85% and above, the time needed to find a 6-bit string of free bits drops. By an MTU size of 16 pages, this effect is fully apparent. When bitmaps are more than 50% allocated, fragmentation drops because of string coalescing, and performance improves because the bitmap search algorithm requires less time in the inner loop. This effect dominates for MTUs longer than 5 pages. At 16 pages and beyond, the bottom half of the plot appears to be lifted by the upper half so that by 256 pages, the two fall in line. This outcome points out that the rise in time associated with increasingly difficult searches due to rising fragmentation follows the same trend (albeit in an opposite direction) as the time savings that is gained from decreasing fragmentation due to string coalescing. At the 256 page mark, this relationship is fixed; for increasingly larger MTUs, the curve remains unchanged except that it rises uniformly in time.

4.3 Performance Comparisons

Three data sets were extracted from the *random* benchmarks. As mentioned in Section 2.5, a program was used to filter out the absolute maximum and minimum times for each MTU size. Also, all iterations that fell in the 40% (±1%) and 10% (±1%) allocation levels were averaged and plotted per message size. The former range was selected because it represented the lowest percentile curve (in 10% increments) that outperformed *memcopy* for all MTU

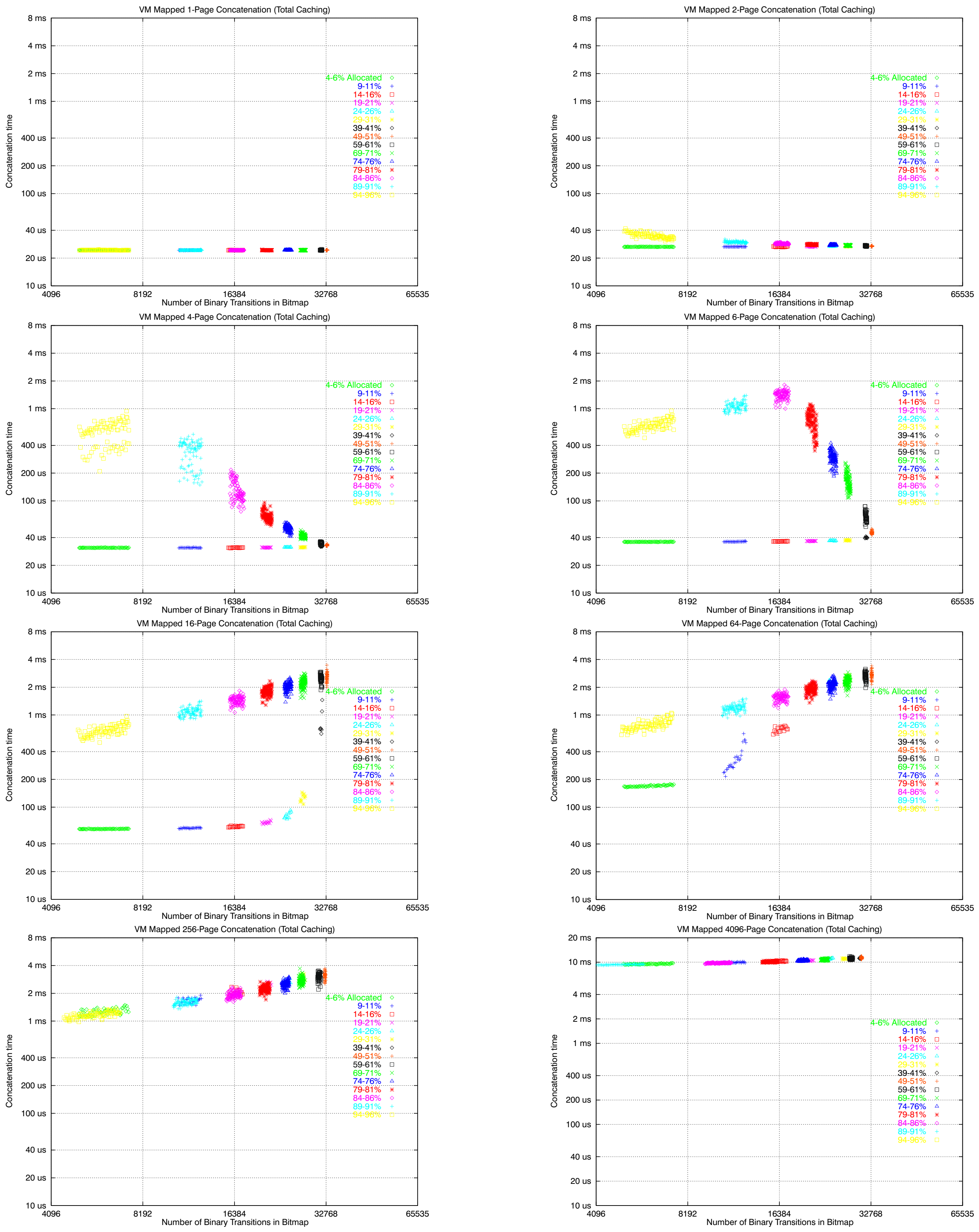

Fig. 7. *Random* Test Results

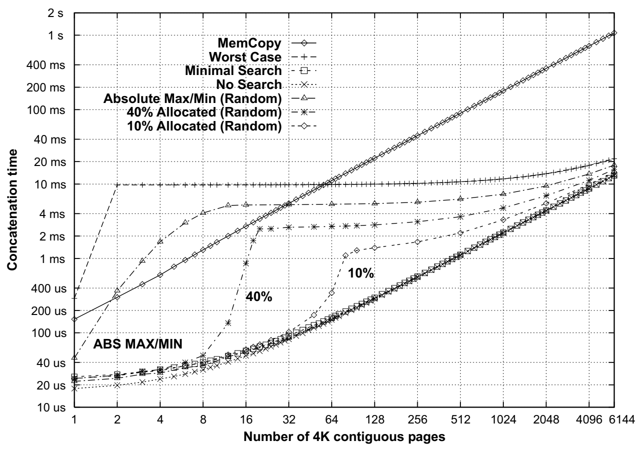

Fig. 8. *Total Caching* vs. Selected *Random* Tests

lengths. The latter range was chosen to gauge low VMMR usage (again in 10% increments) against heavy bitmap use (50% allocation or more). The results are shown overlaid with the *total caching* data in Figure 8. Two important observations can be inferred from this plot. First, absolute maximum times from the *random* studies are significantly better than the timings collected from the *worst case* test; however, for MTUs ranging from 2 to 32 pages, *memcopy* still holds a performance advantage over the VMMR bitmap implementation. Secondly, if the virtual page pool bitmap is 40% allocated or better, VMMR always outperforms standard memcopy. For those applications that do not need more than 102 MB (40% of a 256 MB segment) of buffer space, the VMMR technique, as it has been presented so far, can be extremely useful in maximizing overall network performance.

We can optimize the VMMR bitmap implementation, however, to provide one to two orders-of-magnitude performance improvement over memcopy by simply assuming that all VMMR data is of the same size. Therefore, each bit in the bitmap no longer represents a single 4 KB page, but rather a set of pre-concatenated pages of length equal to that of the message requiring reassembly. All *random* test results will then resemble those of the one page MTU plot shown in Figure 7 (shifted in concatenation time by some MTU-size dependent amount) with all data points falling along the *minimal search* curve in Figure 4. This happens because the optimized algorithm only needs to search for one bit in the map; PTE allocation time per MTU size remains the same, however, thus contributing to the time shift. If certain ranges of MTUs are required, this optimized VMMR bitmap technique could be generalized to handle multiple appropriately sized maps. Note that, unlike reassembly solutions which force a communications card to DMA data into a preallocated contiguous buffer in physical memory, the optimized VMMR scheme does preallocation in virtual memory and allows data to be DMAed into noncontiguous physical pages.

Figure 9 shows the *random test* results redone using the optimized VMMR bitmap technique with a map optimized for 16 page MTUs. Note that the curve resembles the flat one page MTU plot in Figure 7 but is offset by the time required to allocate the VMMR PTEs in the hardware page table. Interpolated results of an optimized VMMR bitmap worst case are overlaid with the unoptimized *total caching* results in Figure 10. These numbers were obtained by partitioning *minimal search* (*total caching* category) reassembly times, per MTU size, into bitmap search times and PTE allocation times. The first numbers were then divided by their corresponding MTU page sizes (the optimized VMMR bitmap length is a scaled version of the original 8 KB bitmap) and added to their PTE allocation times. Note from Figure 10 that, despite optimization, memcopy still outperforms worst case VMMR at the one page MTU size. This reaffirms previous statements noting that a bitmap solution is not an ideal one for VMMR; organizations employing hashing or binary searches will generally always outperform the linear search algorithm described in this paper. Future VMMR studies, then, should concentrate on finding optimal page pool organizations for popular OS and architecture combinations.

4.4 Operating System Implementation Considerations

The results presented thus far were taken from a software environment based on the NAP kernel; certain design considerations should be taken into account during development of a VMMR scheme for an operating system. These include the following:

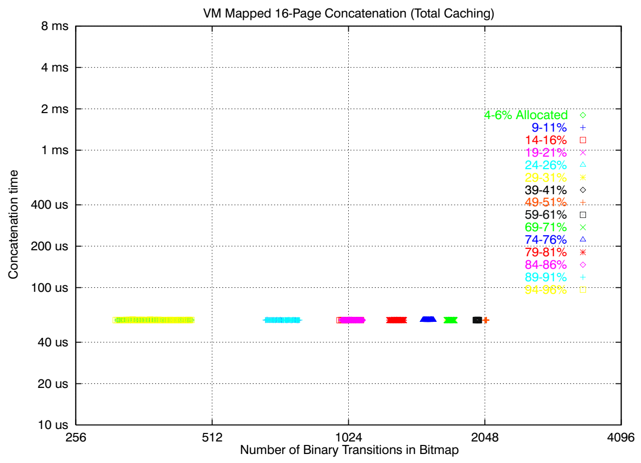

Fig. 9. Optimized VMMR *Random* Test

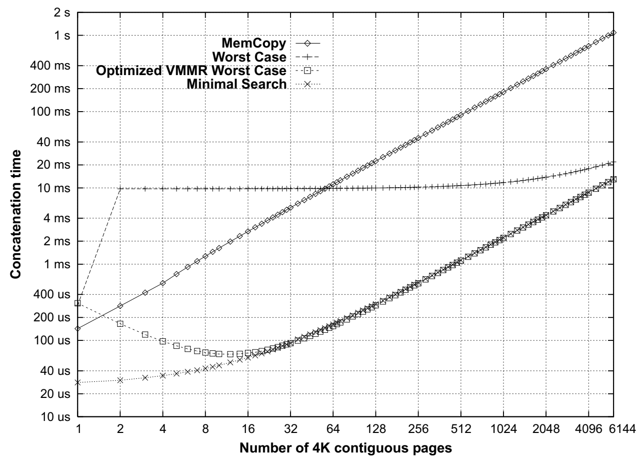

Fig. 10. Optimized VMMR Worst Case

1. Since the VMMR segment is shared among user processes, security mechanisms protecting data from unauthorized users are warranted.

2. After MTU reassembly, a vector of *N* continuous virtual addresses is used to access the newly translated physical pages. Therefore, the communications API does not have to invoke special protocols to implement a true zero copy. In cases where data copy is required (e.g. copy-on-write), standard copy calls should invoke bcopy (continuous byte copy) for improved efficiency.

3. The creation of a virtually reassembled MTU is done in the kernel, so no VM lock operations are needed. Once the MTU has been reassembled, updates to the page table are short enough to be performed in an uninterruptible manner and do not have to be guarded by VM locks as well.

4. VMMR is fully integrated with the VM hardware and software subsystems and is compatible with kernel performance optimizations such as lazy evaluation and copy-on-write.

5. Page wiring can be avoided by removing I/O pages from the kernel's free page list; this prevents the swapper from removing or replacing them during and after data transfers.

6. VMMR can also enhance performance of distributed systems, such as:

Multimedia servers. These systems often receive large messages from several different sources. Traditional buffering systems like those mentioned in the introduction are difficult to manage in such cases.

Network of Workstations (NOW). This system requires very fast user-to-user communications in order to synchronize processes running on different machines.

5. Summary and Conclusions

This paper studies the feasibility of reassembling MTUs using existing virtual memory hardware. A Virtual Memory MTU Reassembly (VMMR) method is compared to an aggressive memory copy (memcopy) benchmark to characterize and evaluate VMMR for use in the PowerPC architecture. We found that VMMR is nearly one to two orders of magnitude faster (up to 14.7 Gbits/sec for a message size of 6,144 pages) than a standard memcopy call (176.9 Mbits/sec).

The studies of Section 4.1 show that in a best case scenario, VMMR offers a performance improvement over memcopy of approximately 5x (28 ms vs. 143 ms) and 83x (13.1 ms vs. 1.09 s) for MTU sizes of 1 (4 KB) and 6,144 (24 MB) pages, respectively. Since the VMMR implementation used in this research (bitmap based free page pool) linearizes its search for available address areas, we also examined a worst case scenario and found the technique to be slower than memcopy for MTU sizes of 1 to 64 pages. Experimental results for L1 cached and non-cached environments showed that both VMMR and memcopy experience significant performance improvements in a cached system, as expected.

Section 4.4 demonstrated that a VMMR bitmap implementation is sensitive to the amount of time required to search a bitmap pool of free addresses for some contiguous block of virtual memory. To quantify this sensitivity in typical operating environments, we examined reassembly times when the virtual page pool bitmap was randomly allocated. The non-intuitive results of Figures 7 and 8 show that VMMR is guaranteed to outperform memcopy as long as the bitmap is less than 40% allocated. In Section 4.5, we discussed how a VMMR bitmap implementation's performance can be further improved if the messages requiring reassembly are fixed in size. Typical designs could include bitmaps of sizes categorized as *tiny*, *small*, *medium*, *large*, and *huge* that correspond to page sizes of 1, 4, 16, 64, and 256, respectively.

We will incorporate VMMR in future work to reassemble MTUs transmitted from a modern, high-performance ATM network interface (see **[16]** for details). Digital video conferencing using 30 fps transmission can take advantage of such a technique. In this system, each digitally encoded frame can be contained within a single MTU for transport across the network. The network interface hardware then streams the MTU into system memory in multiple, discontiguous hardware pages. Each frame is subsequently reassembled at the transport layer interface using VMMR.

Acknowledgments

The authors wish to express their gratitude to the many individuals who helped provide resources and information to allow this work to progress to completion. Brent Paulovicks for his help with the NAP port and his invaluable insights into PPC 601 assembly programming. Dickson Lee and Matthew Thoennes for their system maintenance efforts. Maria Butrico of IBM T.J Watson Research and Ron Minnich of the David Sarnoff Research Center for their participation in several informal discussions on virtual memory mapping.

References

[1] ATM Forum. *ATM User-Network Interface (UNI) Specification*. Version 3.1. Prentice-Hall, 1995.

[2] Traw, C. Brendan S. and J. M. Smith. "Hardware/Software Organization of a High-Performance ATM Host Interface." *IEEE Journal on Selected Areas in Communications*, Vol. 11, No. 2, Feb. 1993.

[3] Clark, David D., Van Jacobson, John Romkey, and Howard Salwen. "An Analysis of TCP Processing Overhead." *IEEE Communications Magazine*. Jun. 1989.

[4] Leffler, Samuel J., Marshall K. McKusick, Micheal J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, 1989.

[5] Watson, R. W. and S. A. Mamrak. "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices." *ACM Transactions on Computer Systems*, Vol. 5, No. 2, May 1987.

[6] IDT. *IDT77201 NICStAR™ PCI ATM Controller Programming Manual*. Version 1.0, 1995.

[7] IBM Microelectronics. *PowerPC 601 System Design Kit: Vol. 1*. May 1994.

[8] Motorola Inc. *PowerPC 601 RISC Microprocessor User's Manual*. Revision 1, Oct. 1993.

[9] Knuth, Donald E. *The Art of Computer Programming: Vol. 2 – Seminumerical Algorithms*. 2d ed. Addison Wesley, 1981.

[10] Thadani, Moti N. and Yousef A. Khalidi. "An Efficient Zero-Copy I/O Framework for UNIX." Sun Microsystems Laboratories, Inc. Technical Report TR-95-39. May 1995.

[11] Tzou, Shin-Yuan and David P. Anderson. "The Performance of Message-passing using Restricted Virtual Memory Remapping." *Software – Practice and Experience*, Vol. 21, No. 3, Mar. 1991.

[12] Druschel, Peter and Larry L. Peterson. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility." *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Dec. 1993.

[13] Chu, Hsiao-keng Jerry. "Zero-Copy TCP in Solaris." 1996 USENIX Annual Technical Conference. Jan. 22-26, 1996. San Diego, CA.

[14] http://www.sun.com/sparc/stp2220/datasheets/01.html

[15] Sedgewick, Robert. *Algorithms in C*. Addison-Wesley, 1990.

[16] Mraz, Ronald, Douglas Freimuth, Edward Nowicki, and Gabriel Silberman. "Using Commodity Networks for Distributed Computing Research." IBM Research Report RC 20445 (89661).