M.I.T. Laboratory for Computer Science 9 February 1977

Computer Systems Research Division Request for Comments No. 135

FURTHER RESULTS WITH MULTI-PROCESS PAGE CONTROL

by R. F. Mabee

This memo updates performance measurements reported by Andy Huber in his recent thesis "A Multi-process Design of a Paging System", now available as MAC-TR 171. The PL/I code is brought up to date with NSS, and improved by removing many external subroutine calls from the critical page fault paths. This gives a performance improvement of about 30%. Many detailed measurements have been made; the results are used to determine where time is spent in both this and the standard page control.

This should be the final report on this project, as no further development is expected.

I. Review

In one chapter of his thesis, "A Multi-process Design of a Paging System", Andy Huber reports measurements made on two versions of Multics, one using his multi-process page control (MPPC) and the other using the standard page control. The former has two H-procs (fast system processes) that run the resource freeing functions of page control, and perform some operations for segment control (typically truncating a page table). Most of the code was rewritten in PL/I, except for the bulk store DIM, a piece of the fault handler, and the system interrupt handler, which are essentially unchanged. The results show comparatively poor performance by the MPPC in two respects:

1) The number of page faults (during a standardized benchmark run) is much higher.
2) The CPU time spent by the PC processes is excessive, doubling the time per page fault.

The increase in page faults can be attributed to the reduced size of the paging pool. The wired stacks, the RWS buffer, the increased size of the PL/I code, and the free core list reduce the paging pool by 10 to 20 pages. This could be cut in half by careful tuning of the algorithm, and becomes unimportant in systems with larger memory. Huber also points out that MPPC disconnects pages before writing them, while the standard PC leaves modified pages connected for an extra lap. If modified pages are more likely to be referenced than unmodified pages, then the standard PC will have fewer page faults.

The increased paging isn't very interesting, because it's readily explained and wouldn't much matter in more reasonable configurations.

For comparisons of CPU time, we adjust the sizes of the paging pools so that the metering run takes about the same number of page faults with each version of PC.

There are two special processes in MPPC: the core manager and the paging device (PD) manager. They perform functions that are mostly done at page fault time in the standard PC, so the MPPC should spend much less time in the page fault handler. Instead, the time is slightly higher (3%). This is the effect of using PL/I. Huber predicts a 40% improvement by replacing external calls with internal calls, with the resulting times shown in the last column of the table.

	Standard PC	MPPC	Predicted
Page fault handler	1973	2043	1226
PC processes	--	2641	1585

Table I. usec per fault. Adapted from Huber.

Three modifications should be made to these numbers for more accurate comparison. In both versions of PC, the fault time meter is updated about 500 usec too soon, before the bulk store read (if any) is posted. There is no question that the time should be accounted to the page fault handler; it's just a bug. Also, the time spent by the PC processes on operations other than page faults (primarily truncation) should be subtracted from the totals; by reasonable extrapolation from more recent measurements this amounts to 336 usec per fault. Thirdly, the cost of interrupt handling and of inter-process swapping (getwork time) should be included; again, these numbers are taken from recent runs. The corrected figures appear in the next table. Comparing the total times, we find MPPC just under twice as expensive.

	Standard PC	MPPC	Predicted
Page fault handler	2473	2543	1726
PC processes	--	2305	1383
Interrupts and getwork	445	684	684
	2918	5532	3824

Table II. usec per fault. Approximate corrections added.

II. Recent changes

For this new series of experiments I used version 28-10 of Multics, with both standard and MP page control subsystems. Among other changes since Huber's experiments was the introduction of NSS (New Storage System), with many consequent effects in page control. NSS resulted in a 200 usec improvement in page fault times for the standard PC, although no corresponding improvement was observed in MPPC. I believe this shows the benefit of the long, careful tuning process applied to standard PC; MPPC must compete without such tuning.

Page faults in the IPC benchmark have increased by 10% during this time, probably due mostly to online changes and only somewhat to reduced paging pool. As before, timing measurements are made with paging pools adjusted so the two versions of PC handle about the same number of faults during a standard metering run.

The final version of MPPC is optimized by embedding subroutines as internal procedures of the page_fault and core_manager programs so that most external calls and redundant assignments (i.e. "sstp = addr (sst$);") are avoided. If all of the external calls could have been removed, then the predictions in Table II would be realized. However, the calls to ALM subroutines (such as the bulk store DIM) couldn't be removed. Moreover, some of the calls that Huber counted to make his

predictions are executed only once in several page faults; in that case the cost per fault is proportionally lower, reducing possible optimization.

Six external calls were removed from page_fault, leaving only four calls, all involving ALM. Seven external calls were removed from core_manager, leaving four to or from ALM. However, three of the calls removed were executed only half the time (when a page must be written). If each external call costs 70 usec, the net gain is only 800 usec, or 14%. The rarer cases aren't optimized, on the grounds that a small improvement in an unusual case wouldn't affect the average times very much. Specifically, only PD reads, page creations, virtual writes, and PD writes not requiring PD allocation are optimized. This handles 84% of the cases.

As another optimization, the core_manager page removal algorithm is made more efficient, although complex, by starting writes for several pages before waiting on any. The overall results are shown in Table III.

	28-10 Standard	Original MPPC	Predicted by Huber	Observed by me
Fault handler	2531	2543	1756	2162
Core manager	--	1985	1191	1272
PD manager	--	320	192	312
Interrupts and getwork	445	684	684	684
	2976	5532	3823	4430

Table III. usec per fault. Results of optimizations.

III. Where the time goes

It is possible to attribute the total CPU time spent on a page fault to the various functions performed. The bulk store DIM alone accounts for about 500 usec per read or write in both systems, which

is surprisingly high. This apparently indicates that the I/O greatly slows the CPU by competing for memory cycles. Of course, this behavior should be unique to the test configuration combining MOS memory with bulk store. Depending on whether the CPU is locked out entirely or just slowed down, this effect may also be slowing down the rest of PC. Another 500 usec is spent (mostly by page$done) to report completion of the I/O. In the following table, the measured time for the standard PC page_fault is arbitrarily divided between freeing core and real page_fault in the proportion measured for the MPPC system. The unusual cases of page creation or forced write to disk are ignored.

	28-10 us/event	28-10 us/fault	MPPC us/event	MPPC us/fault
Real page_fault	482	482	1162	1162
Getwork awaiting core	--	--	637	54
DIM and page$done	1000	1000	1000	1000
Getwork awaiting disk	692	69	637	64
Interrupts, disk read	1921	192	2102	210
Getwork for pre-empt	692	69	637	50
Freeing core frame	297	297	715	715
DIM if must write	1000	557	1000	557
Getwork by core_manager	--	--	637	124
Freeing PD record	580	83	1400	200
DIM if must RWS	2000	112	2000	112
Getwork by pd_manager	--	--	637	56
Interrupts, RWS	1921	115	2102	126
		2976		4430

Table IV. Detailed breakdown of page fault cost.

The total CPU time per fault for MPPC is 1454 usec longer, or about 49%. Approximately 230 usec of the excess is spent in getwork when any process has to wait for a PC process to refill some free list, or when the PC process is done and goes to sleep. Perhaps an

equal amount (unmeasured) is spent in calls to perform the inter-process communication required for the PC processes. An estimated 300 usec represents the effect of less common paths that I didn't bother to optimize, and the cost of putting free frames on a separate list, and the cost of the extra metering done in this version. The rest of the excess (estimated at 700 usec) is directly caused by using PL/I to express the algorithms, which apparently increases the execution time of comparable operations by about 80%. (Note that Huber chose PL/I for ease of implementation, and not for performance.)

One important factor adding to the cost of PL/I is the frequent use of the pointer built-in function (to follow the many threads used by PC). In the ALM version this is done by one instruction, loading an index register. The PL/I compiler optimizes to shorten the generated code; this is not always best for execution speed. Furthermore, the ALM version optimizes register usage over a much larger scope. Mostly these are problems inherent in the use of PL/I, so (unless some gross bug is found) the best performance that might be achieved must still be 20% poorer (in total CPU time per fault) than the standard PC. It's worth noting that the interrupt times for MPPC are only slightly higher (181 usec). The system interrupt handler and disk DIM (both unchanged) use most of the time; the difference is in page$done, a very short procedure converted to PL/I for MPPC. Its execution time is around 400 usec, so the 80% PL/I overhead is still consistent.

In the test configuration, the page fault rate is somewhat less than 100 per second. Since the excess time for MPPC is 1454 usec per

fault, it should cost less than 145400 usec per second, or only 14% of the elapsed time for any run. However, overall system performance is not that much worse. In fact, the faulting process is delayed 369 usec less by the fault (from Table III), so it seems to run faster, and can respond to interactions faster (if it needs only a few new pages).

The PC processes sometimes run during time that would otherwise be idle. The benchmark results show this effect clearly if the working set estimator is enabled -- that reduces multiprogramming and increases idle time, so the MPPC system completes the benchmark in just 8% more elapsed time. (Tuning parameters: WSF = 1, Max Elig = 4; about 150 pages; 23% idle with standard PC.) The MPPC will provide faster service than the standard PC if there is enough idle time. If the PC processes always take what would otherwise be idle time, the page fault costs 369 usec less; if they never do, the fault costs 1454 usec more. At a point in between, the extra cost of MPPC is zero; this happens if the PC processes take idle time 80% of the time. Thus MPPC performs better than the standard PC if there is at least 80% idle time.

The paging function is exercised so heavily in the tiny test configuration that its cost is exaggerated in importance. A system with much larger main memory and no bulk store, which seems to be the right approach for Multics, might, for example, take only ten page faults per second per CPU. In this environment MPPC (minus the PD process) would cost only 4% of the total time, versus 2.8% for the standard PC. The reduction in the paging pool caused by maintaining a free list (in MPPC) would also be unimportant in such a configuration.

Since choosing the right page to evict would become relatively more important than doing it fast, alternative strategies should be tried, and for such experiments the modularity, readability, and PL/I-ness of MPPC make it ideal.

IV. Conclusions

First, the negative recommendations: MPPC as coded is not suitable for installation on a thrashing system like MIT-Multics. It is not ready for use anywhere because of glossed-over NSS issues, incomplete error handling, and just plain bugs. I have no intention of updating the code to more recent Multics releases than 28-10.

There are many positive results. The cost of the inter-process communication and swapping is not too bad (400 usec per fault?), and it could be made much lower by making the free lists longer. (The measurement runs were made with a maximum of 12 free cmes on the list. Because of the interaction with paging rate this size free list would be used only with paging pools from 500-1000.) The delay seen by a process when it faults is slightly reduced. The PL/I version of page control is available as a better base for experimentation and metering than the ALM version.

It turns out that the cost of using general-purpose processes and inter-process communication facilities, while small, is intrinsic. This cost would probably not be much reduced using another implementation of the process, such as Dave Reed's Virtual Processor, since a lot of the cost is in unavoidable overhead of process switching or of calls to perform IPC. Many of the IPC operations either implement a cross-process call to a specific routine, or merely

indicate that (say) the core_manager should be run sometime soon to free up more core frames. The latter function could be more cheaply implemented, at the expense of modularity, if the scheduler called the core_manager directly just before going idle. Of course, if the core_manager isn't a real process, it loses the ability to wait on I/O or on a lock.

By far and away, the biggest performance problem is the use of PL/I. It has already forced a non-modular design for the main programs, by imposing a stiff penalty for good design; it also handles the list-structured objects of page control very poorly. In order to obtain better performance, I would have to rewrite the programs to use constructs for which the code is known to be particularly good; that means picking out the machine language sequence I want first, then fooling the compiler into emitting it. It just isn't worth writing any program in higher-level language if its performance is so important and the language so poorly suited.

Let us momentarily suspend disbelief, to consider an ALM version of MPPC. It should execute similar functions at the same speed as the standard PC, so the extra cost is just the 400 usec presumed for IPC and swapping, or only an 8% increase in CPU time per fault. The delay at fault time becomes 1049 usec less (from Table IV), so overall performance is improved for any load up to 72% (i.e. more than 28% idle). In fact, if the IPC and swapping were optimized as previously suggested, the overall performance might be improved at any realistic load.

Even the ALM MPPC would cause some loss in throughput if there were no otherwise-idle time to give to the PC processes. In the face

of strong real-world emphasis on execution speed, it's sometimes hard to explain why the program with good organization and modularity, clearly expressed in higher-level language, is better than its assembly language predecessor. We have no way of measuring the intangible benefit of any such improvement or of weighing it against a known cost in CPU cycles or dollars. All we can fall back on is the general philosophy, "Good is better than evil, because it's nicer."