Partial Paging for Real-Time NoC Systems

Adrian McMenamin and Neil C. Audsley
Department of Computer Science, University of York, UK
email: [acm538,neil.audsley]@york.ac.uk

***Abstract*—In multiprocessor Network-on-Chip (NoC) architectures it is common that CPU local memory space is limited, with external memory accessed across the NoC infrastructure. Clearly it is imperative for real-time performance that local memory is used effectively, with code and data moved from external memory when required. One common approach is for the local memory to be comprised of two levels, ie. cache and memory. Software mechanisms are used to move code and data between local memory and external memory, eg. scratchpad mechanisms. In this paper we explore the issue of using paging to supplement this approach, ie. a hardware mechanism to automate movement of code and data between external memory and per-CPU local memory within the NoC. This has wide-ranging potential benefits in from efficiency and real-time performance, through application programmability (ie. potential support of logical address spaces). However, the limited amounts of local memory raise the problem of thrashing. Therefore, we examine the effect of limiting thrashing effects by only loading the parts of pages that are referenced (rather than the entire page). The approach is assessed against a real-time video application, considering different page replacement policies.**

I. Introduction

Both transistor scaling [1] and power density limitations [2] have motivated the move towards multiprocessor architectures. However, it is often not possible to provide the many CPUs within a chip large local memories. In multiprocessor Network-on-Chip (NoC) architectures it is common that CPU local memory space is limited, with external memory accessed across the NoC infrastructure - eg. Tilera [3], Intel SCC [4] and Epiphany [5].

The management of this hierarchical memory architecture efficiently so that real-time performance can be maintained is challenging. We note that this is a historic problem – CPUs speeds have generally increased faster than memory (and bus) speeds, forming a memory bottleneck as systems had to wait excessive times for new code and data to be loaded from slower layers in the memory hierarchy. If management of the memory hierarchy is not sufficient, then the overall architecture will spend more time moving code and data between local and external memory than actually computing – the phenomenon of "thrashing" [6].

The most efficient way of populating this local, faster, memory uses the optimal paging algorithm (OPT) – pages with the longest reuse distance are discarded [7]. OPT is "clairvoyant" as it relies on knowledge of future events. While occasionally this knowledge is available to programmers of embedded devices, a more general solution to the problem of thrashing was demonstrated by Denning's "working set" method, which, relying on the strong tendency of computer programs to show locality of reference in the short-term, stipulates that the most effective practical paging policy will be that which retains in memory those pages referenced in the past within a pre-defined time, called the working set window [8]. In fact, Denning's algorithm has proved to be difficult or impractical to implement, but most general computing devices and operating systems use an approximation, typically some form of "least recently used" (LRU) algorithm.

This paper explores the issue of using paging within NoC architectures. CPUs within the NoC typically have a cache and a small bank of SRAM. Large DRAM banks and permanent storage are available externally, accessed via the NoC mesh [3], [4]. Memory resources on the chip are limited — but time to access external memory is much higher than local memory (partly due to contention over the shared NoC mesh). As a consequence the problem of thrashing reappears. Therefore we examine the effect of limiting thrashing effects by only loading the parts of pages that are referenced (rather than the entire page). The approach is assessed against a real-time video application, considering different page replacement policies.

In section 2 we review relevant related work. In section 3 we model the performance of conventional paging systems. Sections 4 and 5 introduce a new approach where only part of a page is loaded. Section 6 offers a discussion and conclusions.

II. Related Work

The wide variety of parallel programming frameworks is perhaps a testimony to the essential difficulty of programming parallel systems. The problems, such as the limitation imposed by the need for at least some code to be serial - "Amdahl's Law" [9] - as well as the difficulties of maintaining coherence and efficiency across a large number of centres of execution are familiar. They are joined by the need to master a novel technology when considering NoC systems. As the authors of [10] state, it has been difficult to "make it easy to write programs that execute efficiently on highly parallel computing systems." Perhaps this is one reason why research has tended to concentrate on the use of NoCs as specialist accelerators [11]. This is also true of researchers' discussions of virtual memory use on NoCs. For instance, in [12] the authors discuss an efficient caching scheme to accelerate sorting.

Other researchers have examined how memory management for GPUs, which, while being "single instruction, multiple data" devices unlike the "multiple instruction, multiple data" devices we are considering, have much in common with NoCs. In [13] it is noted that OPT is not, in fact, optimal when the size of the working set of the data is much greater than the available local memory capacity. In [14] a method of improving cache performance by dynamically altering memory reuse distance is discussed.

Recent research into paging systems has concentrated on large memory systems. While, in [15], it was shown that smaller page sizes could reduce the fault count, more recent research, such as [16], has emphasised that, with large quantities of physical memory (relatively) cheaply available, minimising the cost of translation between virtual and physical addresses larger page sizes are better options to speed up computing in common use domains.

In [17] alternatives to traditional hardware designs to support virtual memory are explored and a model proposed that saves power and adds flexibility to operating system design.

III. Modelling the Performance of Paging Systems

Standard paging approaches move whole pages of code and data *en bloc* the memory hierarchy. This allows a logical address space to be presented to the application programmer – the familiar abstraction of a single and unified address space. However, this is not common within real-time systems (and largely unsupported on existing NoCs). The remainder of this section considers a standard real-time application and assesses its performance with respect to paging.

The x264 program from the PARSEC benchmark suite [18] was used. It was configured to run with a maximum of 16 threads (as we proposed to model a system with 16 cores) – note 18 threads in total were created, though simulations run no more than 16 at once.

Running the benchmark under a modified version of the Valgrind Lackey program [19], we could separate the memory references of each thread of execution and classify every such reference as one of the following:

- *instruction* – like a *load* sees a memory location is accessed but not modified;
- *store* – where a memory location is written to;
- *modify* – a location is first accessed and then written to in a single interaction.

Whilst every *instruction* has an initial impact similar to that of a *load* (in that the address of the instruction itself must be accessed), an instruction may also cause consequent *loads*, *stores* or *modifies*. Additionally, the point at which each new thread was released was marked.

The modified Lackey program produced an XML stream recording every memory access by every thread in time order. This is then used to model different models of on- and off-chip memory interaction and storage. The XML stream recorded the order in which memory addresses were accessed by each thread but contained no specific timing information and thus did not record any delays for thread synchronisation - so by its nature any processing of the XML could only be an approximation of how different paging policies would behave.

The modelled hardware system has 16 cores, each with 32KB of local memory (forming a 512KB pool of on-chip memory), this was loosely based on the Tilera example [11]. We assumed that all on-chip memory was immediately (i.e., in one "tick") available to all cores (i.e., we ignored both the issues of on-chip synchronisation and on-chip communication delays) and assumed that a standard cache line of external memory (128 bits, or 16 bytes) was available after a delay of 100 cycles/ticks. So, for instance, a 4KB page would take 25,600 ticks to load. The experiment does not model caching behaviour or the costs of writing-back modified pages as these aspects do affect the broad behaviour of the NoC model when using paged memory.

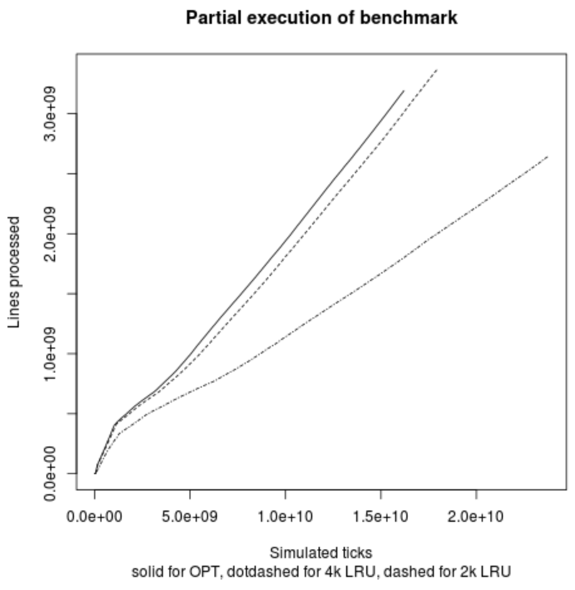

Figure 1. OPT and LRU compared

Our central finding was that FIFO, LRU (including LRU 2Q varieties) and even OPT replacement policies all showed the characteristics of thrashing as the system became memory I/O bound. Additional CPUs did not speed the system up, rather slowing each individual CPU as they were constrained by the small overall pool of memory[1].

Figure 1 shows the simulated performance of OPT and LRU for 4KB pages and also the performance of an LRU algorithm with 2KB page sizes[2]. The number of lines processed indicates progress in completing the benchmark, while the simulated ticks is an analogue for time. It will be seen that although using 2KB page sizes increases performance (despite resource restraints), all the lines, including that for OPT, display a common characteristic - that the rate of progress becomes constant. As Figure 2 shows, applying more CPUs to the task does not speed up its execution: the lines processed per simulated tick remaining constant even as more threads are being executed and more processors are being used. The graph shows that the simulated system is memory I/O bound: additional CPUs cannot squeeze any more computing power from the system as they simply fight each other for access to the limited memory pool.

[1]The model employed barrier synchronisation and if two threads both requested the same page both would gain access to it when it loaded on the earliest request. Threads simply blocked when waiting.

[2]To compensate for the additional size and cost of page tables that 2KB pages would require we allocated 30KB per core and increased the access time to 2 ticks for a present page.

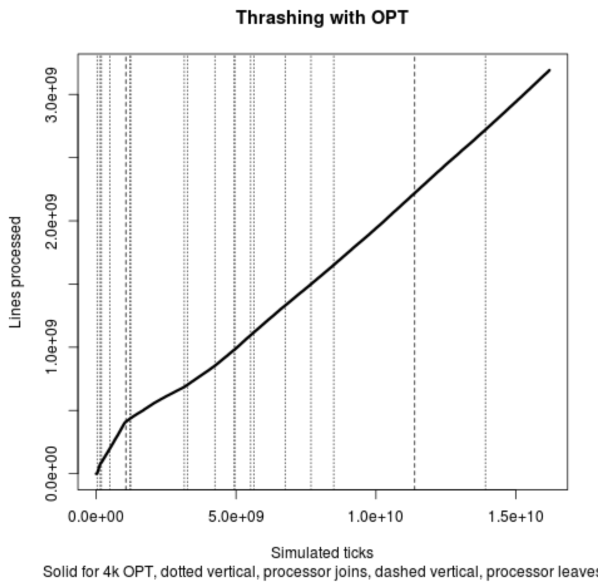

Figure 2. OPT algorithm: more processors do not speed execution

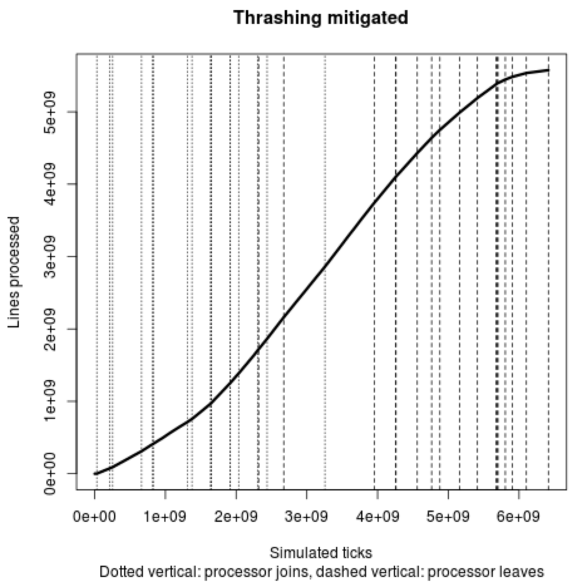

Figure 4. Partial paging: additional processors speed execution

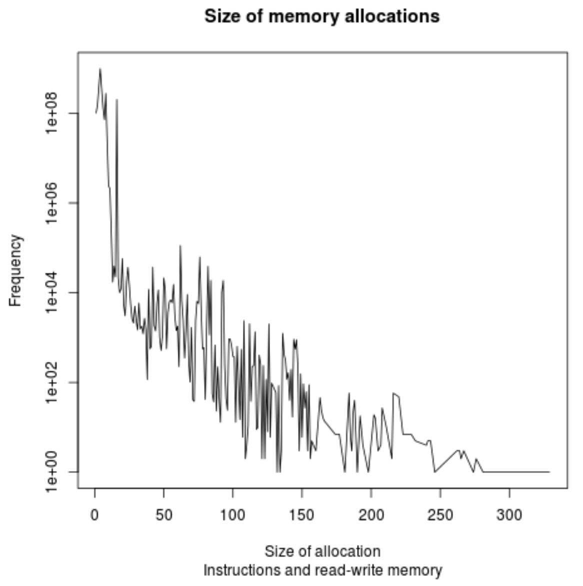

Figure 3. Logarithmic plot of the frequency of different sizes of contiguous memory allocations

IV. Partial Paging Approach

Figure 3 shows small (16 bytes or fewer) contiguous memory allocations were orders of magnitude more likely than larger allocations. Since pages were being pushed out quickly, we tested the proposition that a **partial paging allocation policy** – pages are populated one cache line (ie.16 bytes) at a time – could improve performance.

In this case we used 2KB pages and 30KB per core, with a cost of four ticks to access a present memory block and we tracked whether a given 16 byte block was present through a bitmap. The result, seen in Figure 4, was improved performance: as more threads are executed and additional CPUs used, the processing rate increases – mitigating thrashing.

A. Testing the Partial Paging Approach

The partial paging approach was tested using the OVPSim instruction accurate simulator [20] with MicroBlaze soft CPU [21] which delivers one instruction per cycle, enabling instruction count to be a good approximate to cycle counting.

1) Unmodified Microblaze: Each thread's XML output from the modified Valgrind Lackey was converted into MicroBlaze memory load and write instructions and was executed using simple page tables. In an unmodified MicroBlaze such code will continue to run (assuming no other problems) so long as a translation lookaside buffer (TLB) is able to translate the virtual address being accessed into a physical address. If address was not translatable by a TLB then an exception would be raised – ie. when the memory being accessed is not available "locally" (as though in the on-chip pool) and so must be copied from a "remote" address.

Three TLB entries were "pinned" (ie. made permanent and unchangeable), so ensuring the code providing basic VM services and the generated code, the page tables and the page frames would always have appropriate translations.

The system was configurable, eg. to have more page frames of physical memory than TLB entries. However within this paper we focus on the case where the number of page frames of physical memory was the same as the number of TLB entries (up to the maximum supported 64 TLB entries). In this case every TLB miss corresponds to a "hard fault" – ie. it requires a new page to be loaded into physical memory and, in all cases after the system has used all available physical memory, the eviction of a currently present page[3].

The demand paging FIFO page replacement system was tested to determine the fault count of 4KB and 1KB pages (the two smallest sizes supported on the MicroBlaze). As can

[3]The MicroBlaze has no timing device within OVPSim with so eviction policies followed a "first-in, first-out" (FIFO) policy as opposed to the more efficient CLOCK-type LRU approach

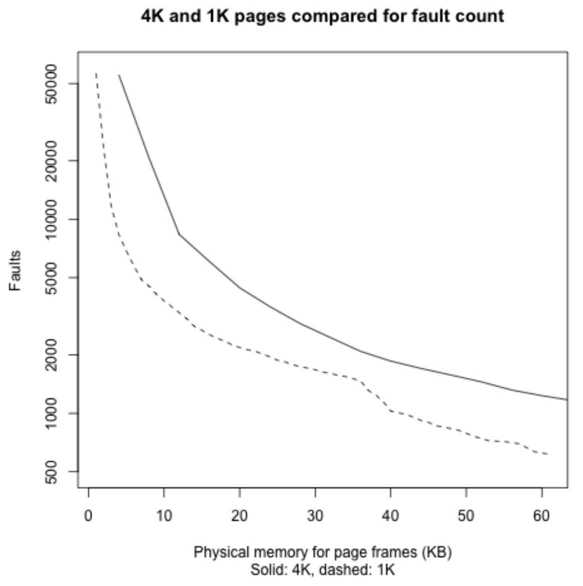

Figure 5. Fault count for traditional paging approach for different page sizes

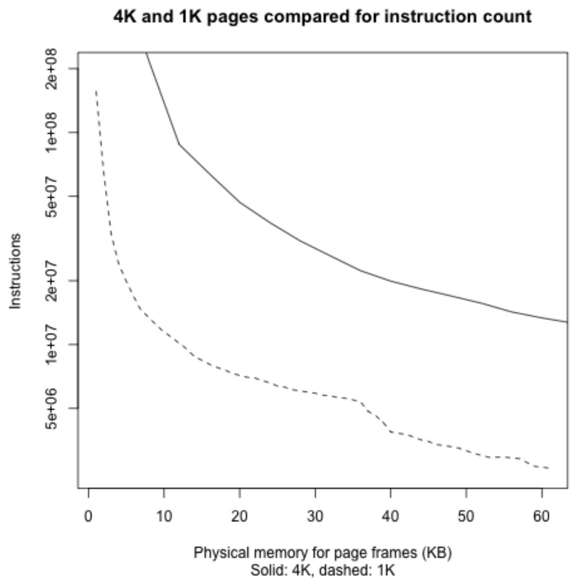

Figure 6. Instructions required to complete task

be seen in Figures 5 and 6, for a fixed amount of local memory, the 1KB pages delivered a lower fault count and required many fewer instructions to be executed to complete the task.

On each page fault that led to an eviction, as well as executing code to manage the page tables, the system was required to write back an evicted page as well as copy the incoming page into memory designated as holding a "local" page frame - no DMA functions were available on this simple model and so this was all carried out in assembly loops that copied memory from one address to another. As can be seen in Figure 6, this made the 1KB page model substantially more efficient than even the lower fault count along might suggest: there were fewer faults and each cost less to handle. At this point we made no allowance for the cost of transferring

TLBs	Instructions: traditional paging	Instructions: alternative paging
4	157,493,205	n/a
8	20,219,450	18,545,020
12	12,651,719	14,717,082
16	9,930,702	13,457,998
20	8,215,518	12,614,663
24	7,457,021	12,270,902
28	6,844,912	12,079,901
32	6,468,068	11,834,218
36	6,140,900	11,717,928
40	5,329,413	11,558,408
44	4,226,715	10,619,623
48	3,897,005	10,453,064
52	3,651,137	10,315,069
56	3,322,324	10,092,510
60	3,296,123	10,076,433
64	2,991,081	9,910,243

Table I
INSTRUCTION COUNTS FOR "TRADITIONAL" AND "ALTERNATIVE" 1KB PAGING SYSTEMS

memory from a "remote" to a "local" address, merely counting the number of instructions required to execute the copy.

2) Microblaze with Partial Paging: The OVPSim Microblaze code was modified to include partial paging – ie. pages loaded in 16 byte blocks. Now, while a TLB miss exception would be thrown in the normal way if an address translation was not available, each reference to an address mapped to "local" memory would raise an interrupt. The interrupt handler then would check a bitmap to see if the addressed 16 byte block has been loaded from remote memory to local memory. If it has no further action was taken and the interrupt handler returns, if it has not then a "small fault" is raised and the appropriate 16 byte line loaded, bitmap updated, and the interrupt handler returns. This means a substantial code block was executed on every memory reference, though the code executed when the fragment being accessed was present was significantly shorter than when it was missing. Hard faults still occur and in most cases (after the initial period when empty physical pages are being written to) require a page write-back (again, we did this for all pages) as well as a low cost bitmap reinitialisation. In such cases, only those 16 byte lines marked as present are written back. On a hard fault only the initially requested 16 byte block was loaded.

As Table I[4,5] shows, comparisons show higher instruction counts for all but the smallest amounts of available local memory. However instruction counts do not provide a full comparison between the two systems. Although partial paging generally executes more instructions to complete the task, it also loads smaller amounts of memory. Each fault on a 1KB traditional system requires a minimum of a 1KB page load - typically costing somewhere between 4800 cycles (if global memory is 75 cycles "away") and 8000 cycles (if global memory is 125 cycles per 16 byte cache line away).In contrast the alternative system only needs to load those lines it requires.

Partial paging shows superior performance when the timing

[4]For the traditional system three TLBs are pinned so, for instance 16 TLBs leaves 13KB for physical pages, for the alternative system four TLBs are pinned and 16 TLBs leaves 12KB for physical pages

[5]The bitmaps were pinned in memory, so losing a further TLB entry and so the alternative system needs a minimum of 5KB or 5 TLBs

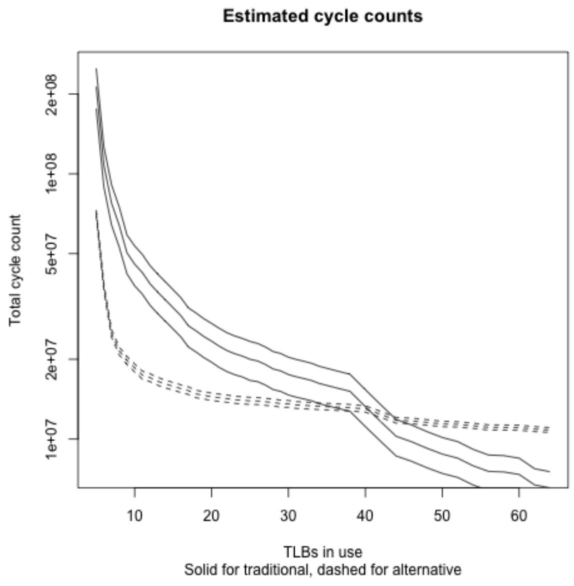

Figure 7. Estimated total cycles required by different paging algorithms: in each case the top line is for global memory 125 cycles away and the bottom 75 cycles away

TLBs in use	(Hard) Faults	Instructions
8	88,875	249,531,853
16	71,404	222,944,264
24	63,276	205,776,575
32	56,527	194,463,472
48	47,217	180,191,027
64	40,905	171,472,116

Table II
FAULT AND INSTRUCTION COUNT RESULTS FOR ILLUSTRATIVE LOW LOCALITY LOAD (TRADITIONAL PAGING)

is normalised. Figure 7 illustrates: the estimated total cycles required if global memory access cost is 75, 100 and 125 cycles per 16 byte cache line is compared for the two algorithms. Here partial paging requires fewer cycles (for this memory access pattern) when local memory is around 32KB or less. The flat performance profile of partial paging suggests this is dominated by the interrupt handler code rather than the number of faults (completing the task requires a set number of memory accesses and so the handler is run a set number of times regardless of the number of TLB entries in use). Improving the performance of this part of the process, such as making the checking of the bitmap a sub-cycle task in hardware could dramatically increase the advantage of the alternative approach.

TLBs in use	Hard and small faults	Instructions
8	113,150	108,389,860
16	112,668	108,556,316
24	112,134	109,586,781
32	111,594	110,708,748
48	110,261	114,716,046
64	108,769	118,125,334

Table III
FAULT AND INSTRUCTION COUNT RESULTS FOR ILLUSTRATIVE LOW LOCALITY LOAD (PARTIAL PAGING)

It should be further noted that, as we did not differentiate between page types[6], we did not account for the cost of writing back pages in this comparison, beyond the instructions required to be executed: such a count would certainly increase the advantage of partial paging. For instance, with 32 TLB entries, the average page has 144 bytes loaded on eviction and so only nine 16 byte blocks would need to be written back. The use of instruction count for comparison does account for the relative complexity of the two situations: in the case of the alternative approach the bitmap must be read to decide which blocks are to be written back.

We further tested the partial approach with a semi-randomised[7] selection of pages and, unsurprisingly, the partial paging approached showed a very strongly enhanced performance, as illustrated in Tables II and III.

V. Potential Additional Adaptations

We were able to consider some additional adaptions to the partial paging algorithm.

A. *Testing other loading sizes*

Partial paging was tested with 32 byte and 64 byte loads. Such larger loads reduce the number of small faults and Table IV summarises the results. The marginal efficiency of the larger loads increases with the amount of TLB entries in use - for 8 TLB entries there are 2.9 more small faults with a 16 byte load size than for a 64 byte load size, while for 32 TLB entries the ratio is 3.1:1 and for 64 it is 3.2:1, but the gains are not dramatic and, given that the number of interrupts raised is the same regardless of the load size used then it is plain that, without hardware adaption, there is no benefit to using larger load sizes.

TLBs in use	Hard faults	Small: 16 bytes	Small: 32 bytes	Small: 64 bytes
8	8357	21122	12612	7375
12	4526	18858	10953	6249
16	3301	17209	9988	5702
20	2543	15822	9105	5203
24	2184	15144	8651	4936
28	1956	14688	8377	4763
32	1741	13893	7876	4472
36	1609	13400	7557	4272
40	1469	12866	7230	4079
44	1027	10623	5983	3367
48	919	10183	5733	3219
64	626	8513	4764	2649

Table IV
FAULT COUNTS FOR DIFFERENT LOAD SIZES COMPARED

B. *Moving from FIFO to LRU*

The presence of an interrupt on every memory access does allow experimentation with an LRU page replacement policy – noting additional costs of management of page lists etc.

[6]We could have assumed that no instruction pages were to be written back but for the sake of simplicity we treated all pages in the same way, so write-back code is executed for all pages

[7]Pages were selected from the same range of addresses and with approximately the same frequency and with allocation sizes modelled on the results shown in Figure 3, but with no stronger bound of locality.

We tested two forms of LRU: a partial policy where the page order was updated only on a hard or small fault, and a full LRU where the page list order was updated on every access. The results are summarised in Table V – both approaches significantly lower the total fault count compared to FIFO. For a 32 TLB system (ie. with 28KB of local memory), there are 9% fewer faults with the partial approach and 25% fewer with the full LRU policy. These would save 142,500 cycles and 388,100 cycles respectively in load time from global memory 100 cycles away. However, the cost of implementing the LRU policies in additional instructions greatly outweigh these, as shown in Table VI. The high cost of manipulating the ordered list decisively counts against the full LRU approach in particular.

TLBs	FIFO	Partial LRU	Full LRU
16	20510	18847	16492
32	15634	14209	11753
48	11102	10337	8355
64	9139	8762	7455

Table V
FAULT COUNTS- HARD AND SMALL COMBINED - FOR DIFFERENT PAGE REPLACEMENT ALGORITHMS

VI. CONCLUSIONS

Virtual memory has been part of the standard programming toolkit for around half a century. In recent years much research focus has been on how to improve the performance of machines with large amounts of memory, yet, at the same time, a problem from the dawn of virtual memory - thrashing - has also reappeared, especially in devices that might be otherwise expected to run highly parallel real time computing tasks, such as video processing, at speed. Our simulations suggest that such systems, if using virtual memory, could improve performance by both using smaller page sizes (and so travel in the opposite direction of systems processing "big data") and adopt a new sub-paging approach of loading in memory in cache line size blocks. However, our initial research also suggests that significant speed improvements will only come if we can match the bitmaps that record which parts of a page have already been populated to accessed addresses in hardware and thus sub-cycle.

We propose that such hardware adaptions would be possible: hardware memory management units (MMU) have long supported address translation and lookup on a sub-cycle basis. We have adopted a bitmap as an efficient method with which to map internal memory allocations in software, but it may be that other methods are more hardware efficient. In [22] a hardware bitmap-based memory allocator is discussed, while [23] discusses an MMU designed specifically for system-on-chip hardware.Further work includes investigation to see if a suitable hardware modification can be made (using an FPGA based software). This can then be used within an existing NoC architecture to evaluate the approach fully.

TLBs	FIFO	Partial LRU	Full LRU
16	13,457,998	15,448,631	31,501,330
32	11,834,218	14,951,934	43,505,073
48	10,453,064	13,852,267	55,552,516
64	9,910,243	13,728,863	68,390,135

Table VI
INSTRUCTIONS EXECUTED FOR EACH PAGE REPLACEMENT ALGORITHM

REFERENCES

[1] Ethan Mollick, "Establishing Moore's Law", *IEEE Ann. Hist. Comput.*, vol. 28, no. 3, pp. 62–75, 2006, 1158837.

[2] M. Bohr, "A 30 year retrospective on Dennard's MOSFET scaling paper", *Solid-State Circuits, IEEE*, vol. 12, no. 1, pp. 11–13, 2007.

[3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect", in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, feb. 2008, pp. 88–598.

[4] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl, "Evaluation and improvements of programming models for the Intel SCC many-core processor", in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, july 2011, pp. 525–532.

[5] Adapteva, "Epiphany Architecture Reference", http://adapteva.com/docs/epiphany_arch_ref.pdf, 2015.

[6] Peter J Denning, "Virtual memory", *ACM Computing Surveys (CSUR)*, vol. 2, no. 3, pp. 153–189, 1970.

[7] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer", *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, June 1966.

[8] P. J. Denning, "Working Sets Past and Present", *IEEE Trans. Softw. Eng.*, vol. 6, no. 1, pp. 64–84, January 1980.

[9] M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era", *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[10] R. Bodik B. C. Catanzaro J. J. Gebis P. Husbands K. Keutzer D. A. Patterson W. L. Plishker J. Shalf S. W. Williams K. Asanovic and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley", Tech. Rep. UCB/EECS- 2006-183, EECS Department, University of California, Berkley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html, December 2006.

[11] D. Ungar and S. Adams, "Hosting an object heap on manycore hardware: an exploration", *SIGPLAN Not.*, vol. 44, no. 12, pp. 99–110, 2009.

[12] Alessandro Morari, Antonino Tumeo, Oreste Villa, Simone Secchi, and Mateo Valero, "Efficient sorting on the tilera manycore architecture", in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*. IEEE, 2012, pp. 171–178.

[13] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt, "Cache-conscious wavefront scheduling", in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 72–83.

[14] X. Chen, L. Chang, C. I Rodrigues, J. Lv, Z. Wang, and W.i Hwu, "Adaptive cache management for energy-efficient gpu computing", in *Microarchitecture Annual IEEE/ACM Int. Symp. on*. IEEE, 2014, pp. 343–355.

[15] Donald J. Hatfield, "Experiments on page size, program access patterns, and virtual memory performance", *IBM Journal of research and development*, vol. 16, no. 1, pp. 58–66, 1972.

[16] A. Basu, J. Gandhi, J. Chang, M. Hill, and M. Swift, "Efficient virtual memory for big memory servers", in *ACM SIGARCH Computer Architecture News*. ACM, 2013, vol. 41, pp. 237–248.

[17] B. Jacob and T. Mudge, "Uniprocessor virtual memory without tlbs", *IEEE Trans. on Computers*, vol. 50, no. 5, pp. 482–499, 2001.

[18] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, "The parsec benchmark suite: Characterization and architectural implications", Tech. Rep. TR-811-08, Princeton University, January 2008.

[19] Nicholas Nethercote and Julian Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", in *ACM Sigplan notices*. ACM, 2007, vol. 42, pp. 89–100.

[20] OVPWorld.org, "Open virtual platforms (ovp) an introduction and overview".

[21] xilinix.com, "Microblaze soft processor core".

[22] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles, "Dynamic storage allocation: A survey and critical review", in *Memory Management*, pp. 1–116. Springer, 1995.

[23] Mohamed Shalan and Vincent J Mooney, "A dynamic memory management unit for embedded real-time system-on-a-chip", in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, 2000, vol. 17, pp. 180–186.