CMCP: A Novel Page Replacement Policy for System Level Hierarchical Memory Management on Many-cores

Balazs Gerofi[†], Akio Shimada[‡], Atsushi Hori[‡], Takagi Masamichi[§], Yutaka Ishikawa[†,‡]
[†]Graduate School of Information Science and Technology, The University of Tokyo, JAPAN
[‡]RIKEN Advanced Institute for Computational Science, Kobe, JAPAN
[§]Green Platform Research Lab, NEC Corp., Tokyo, JAPAN
bgerofi@il.is.s.u-tokyo.ac.jp, a-shimada@riken.jp, ahori@riken.jp, m-takagi@ab.jp.nec.com, ishikawa@is.s.u-tokyo.ac.jp

ABSTRACT

The increasing prevalence of co-processors such as the Intel® Xeon Phi™, has been reshaping the high performance computing (HPC) landscape. The Xeon Phi comes with a large number of power efficient CPU cores, but at the same time, it's a highly memory constraint environment leaving the task of memory management entirely up to application developers. To reduce programming complexity, we are focusing on application transparent, operating system (OS) level hierarchical memory management.

In particular, we first show that state of the art page replacement policies, such as approximations of the least recently used (LRU) policy, are not good candidates for massive many-cores due to their inherent cost of remote translation lookaside buffer (TLB) invalidations, which are inevitable for collecting page usage statistics. The price of concurrent remote TLB invalidations grows rapidly with the number of CPU cores in many-core systems and outpace the benefits of the page replacement algorithm itself. Building upon our previous proposal, per-core Partially Separated Page Tables (PSPT), in this paper we propose *Core-Map Count based Priority* (CMCP) page replacement policy, which exploits the auxiliary knowledge of the number of mapping CPU cores of each page and prioritizes them accordingly. In turn, it can avoid TLB invalidations for page usage statistic purposes altogether. Additionally, we *describe and provide an implementation of the experimental 64kB page support* of the Intel Xeon Phi and reveal some intriguing insights regarding its performance. We evaluate our proposal on various applications and find that CMCP can outperform state of the art page replacement policies by up to 38%. We also show that the choice of appropriate page size depends primarily on the degree of memory constraint in the system.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Storage Management—*Virtual Memory*

Keywords

Page Replacement; Manycore; Xeon Phi

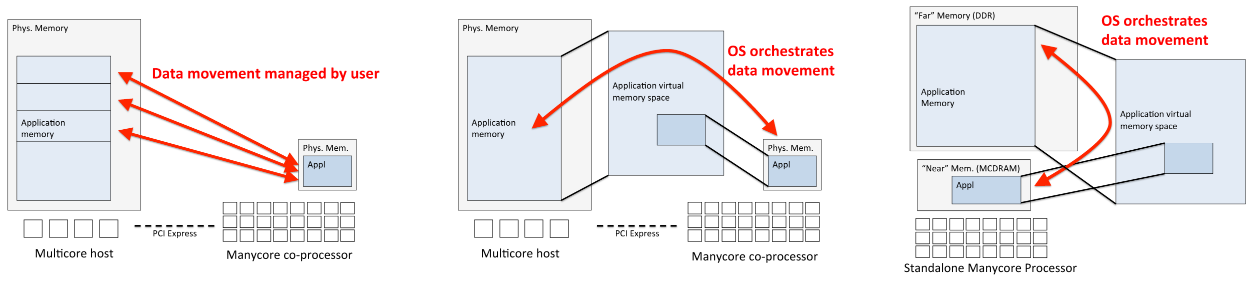
1. INTRODUCTION

Although Moore's Law continues to drive the number of transistors per square mm, the recent stop of frequency and Dennard scaling caused an architectural shift in high-end computing towards hybrid/heterogeneous configurations. At present, a heterogeneous configuration consists of a multicore processor, which implements a handful of complex cores that are optimized for fast single-thread performance, and a manycore unit, which comes with a large number of simpler and slower but much more power-efficient cores that are optimized for throughput-oriented parallel workloads [26].

The Intel® Xeon Phi™ product family, also referred to as Many Integrated Cores (MIC), is Intel's latest design targeted for processing such highly parallel applications. The *Knights Corner* Xeon Phi card, used in this paper, provides a single chip with sixty x86 cores each processor core supporting a multithreading depth of four. Currently, the Intel® Xeon Phi™ comes on a PCI card, and has its own on-board memory, connected to the host memory through PCI DMA operations. The on-board memory is faster than the one in the host, but it is significantly smaller. Only 8 Gigabytes on the card, as opposed to the tens or hundreds of GBs residing in the host machine. This limited on-board memory requires partitioning computational problems into pieces that can fit into the device's RAM and orchestrate data movement along with computation, which at this time is the programmer's responsibility. This architecture with user managed data movement is shown in Figure 1a.

Although current programming models execute applications primarily on the multicore host which in turn offloads highly parallel sections to the co-processor, in the future, focus will shift further towards the manycore unit itself. Intel has already announced details of its next generation Xeon Phi chip, codenamed *Knights Landing*, which will come in a standalone bootable format and will be equipped with multiple levels of memory hierarchy[1] [1], called "near" and "far" memory, respectively. Similarly, Nvidia has also argued that additional levels of memory hierarchies will be necessary for future massively parallel chips [18]. Keeping this architectural direction in mind, this paper primarily focuses on the manycore unit itself and investigates how the host memory can be utilized from the co-processor's point of view.

[1]The term *multiple levels of memory hierarchy* does not refer to multiple levels of caches in this paper.



(a) Manual data movement between the host and manycore co-processor on current heterogeneous architectures. *(Offload model.)*

(b) OS driven data movement between the host and manycore co-processor on current heterogeneous architectures. *(Proposed model.)*

(c) OS driven data movement on future standalone many-core CPUs with multiple levels of memory hierarchy. *(Proposed model.)*

Figure 1: Overview of data movement scenarios on current heterogeneous systems and future standalone manycore CPUs with multiple levels of memory hierarchy.

Because the Intel® Xeon Phi™ co-processor features a standard memory management unit (MMU), it is capable to provide much larger virtual memory than that is physically available. The operating system can keep track of the physical memory, manage the mapping from virtual to physical addresses, and move data between the card and the host in an application transparent fashion. This proposed OS level data movement on current heterogeneous architectures and future standalone many-cores is shown in Figure 1b and Figure 1c, respectively.

We emphasize that data movement is *inevitable* in these architectures and we investigate the feasibility of a system level solution. While OS level data movement may sound analogous to swapping in traditional operating systems, the scenario of current manycore co-processor based memory management is considerably different than regular disk based swapping on a multicore CPU. First, data movement between the co-processor's RAM and the host memory, which takes place through the PCI Express bus, is *significantly faster* than accessing a disk in the host. This makes the relative cost of data movement during page fault handling *much lower* than in a disk based setup. Second, the large number of CPU cores on the co-processor renders the cost of remote TLB invalidations using regular page tables (i.e., shared by all cores) much higher than in a multi-core CPU.

We have already proposed per-core *partially separated page tables* (PSPT) to alleviate the TLB problem of frequent address remappings [14]. Further investigating co-processor based hierarchical memory management, we are focusing on page replacement policies in this paper. We show that state of the art replacement algorithms, such as approximations of the least recently used (LRU) policy, are not good candidates for massive many-cores due to their inherent cost of TLB invalidations required for tracking page usage statistics. LRU based algorithms aim at decreasing the number of page faults by keeping the working set of the application close to the CPU in the memory hierarchy [6]. To approximate the working set, however, they rely heavily on the access bit of page table entries which needs to be checked and cleared periodically. Unfortunately, on x86 each time the access bit is cleared in a PTE, the TLB for the corresponding virtual address needs to be invalidated on all affected CPU cores. We find that despite the fact that LRU successfully decreases the number of page faults, the price of frequent TLB invalidations for monitoring page usage eventually outweighs the benefit of the page replacement algorithm itself.

To address this issue, we propose a novel page replacement policy which relies on the auxiliary knowledge of the number of mapping CPU cores for each address obtained from the per-core page tables. Intuitively, pages that are mapped by a large number of CPU cores are likely more important than those mapped by only a few. Furthermore, swapping out such pages requires TLB invalidations on a large number of CPU cores. Therefore, our algorithm prioritizes victim pages based on the number of mapping CPU cores and in turn it can avoid remote TLB invalidations for page usage statistics altogether. Moreover, we also consider the impact of using different physical page sizes supported by the Xeon Phi. We summarize our contributions as follows:

- Building upon PSPT, we propose a *Core-Map Count based page replacement Policy* (CMCP), which prioritizes victim pages based on the number of mapping CPU cores when pages are moved between the host and the MIC and compare its performance against various page replacement policies.

- We *describe and give an implementation of the experimental 64kB page size feature* of the Xeon Phi (which currently goes unused in Intel's Linux stack [17]) and reveal various insights regarding its performance in the context of OS level hierarchical memory management. To the best of our knowledge, this is the first time the 64kB page support of the Xeon Phi is discussed.

- Additionally, while we presented preliminary results earlier for partially separated page tables on a simple stencil computation kernel using 4kB pages [14], we *further evaluate PSPT (both with and without the CMCP policy) running various NAS Parallel Benchmarks and SCALE*, a climate simulation stencil application developed at RIKEN AICS, including measurements on the *impact of different page sizes*.

We demonstrate that partially separated page tables work well on real applications and provide scalable memory management with the increasing number of CPU cores in contrast to regular page tables, which fail to scale over 24 cores. We also show that the core-map count based replacement policy outperforms both FIFO and LRU on all applications we investigate by up to 38%.

Moreover, we confirm that the optimal page size depends on the degree of memory constraint imposed and demonstrate that under certain circumstances 64kB pages can yield superior performance compared to both 4kB and 2M pages.

The rest of this paper is organized as follows, Section 2 provides background and gives on overview of PSPT, Section 3 discusses the design of core-map count based replacement policy and Section 4 describes the 64kB page support of the Xeon Phi. Section 5 provides experimental results, Section 6 surveys related work, and finally, Section 7 concludes the paper.

2. BACKGROUND

2.1 Interface for Heterogeneous Kernels

The Information Technology Center at the University of Tokyo and RIKEN Advanced Institute of Computational Science (AICS) have been designing and developing a new scalable system software stack for future heterogeneous supercomputers.

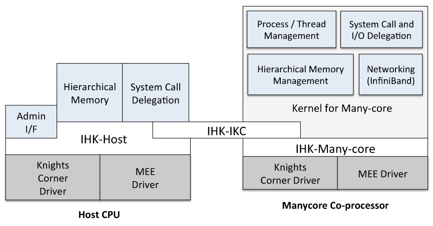


Figure 2: Main components of Interface for Heterogeneous Kernels (IHK) and the manycore kernel.

Figure 2 shows the main components of the current software stack. The *Interface for Heterogeneous Kernels* (IHK) hides hardware-specific functions and provides kernel programming interfaces. One of the main design considerations of IHK is to provide a unified base for rapid prototyping of operating systems targeting future many-core architectures.

IHK on the host is currently implemented as a Linux device driver, while the IHK manycore is a library that needs to be linked against the OS kernel running on the co-processor. Another component of the IHK worth mentioning is the *Inter-Kernel Communication* (IKC) layer that performs data transfer and signal notification between the host and the manycore co-processor.

We have already explored various aspects of a co-processor based system, such as a scalable communication facility with direct data transfer between the co-processors [29], and possible file I/O mechanisms [21]. We are currently developing a lightweight kernel based OS targeting manycore CPUs over the IHK, and at the same time, design considerations of an execution model for future manycore based systems is also undertaken. The minimalistic kernel is built with keeping the following principles in mind. First, on board memory of the co-processor is relatively small, thus, only very necessary services are provided by the kernel. Second, CPU caches are also smaller, therefore, heavy system calls are shipped to and executed on the host. Third, the number of CPU cores on the co-processor board is large so kernel data structures need to be managed in a scalable manner.

2.2 Execution Model

Projections for future exascale configurations suggest that the degree of parallelism inside a node could experience over a hundred fold increase, while the number of nodes in the system will likely grow by at least a factor of ten. In order to realize scalable communication among processes running on such systems, we believe that sharing the address space among multiple CPU cores inside a node, i.e., running a single, multi-threaded process (think of hybrid MPI and OpenMP programming), or at most a few, are the only viable approaches as opposed to assigning separate processes to each core. Thus, our main concern is how to handle a single address space running on a manycore co-processor.

In our current model (shown in Figure 1b) the application executes primarily on the manycore unit (similarly to Intel's native mode execution [16]), but it has transparent access to the memory residing in the host machine, as an additional level in the memory hierarchy. The application virtual address space is primarily maintained by the co-processors and is partially mapped onto the physical memory of the manycore board. However, the rest of the address space is stored in the host memory. The operating system kernel running on the co-processor is responsible for moving data between the host memory and the co-processor's RAM and for updating the virtual address space accordingly. It is worth pointing out that due to the large number of CPU cores the OS needs to be able to handle simultaneously occurring page faults in a scalable manner.

2.3 Per-core Partially Separated Page Tables

In order to provide the basis for further discussion on page replacement policies, this Section will first give a brief overview of our previous proposal, per-core partially separated page tables (PSPT) [14].

With the traditional process model all CPU cores in an address space refer to the same set of page tables and TLB invalidation is done by means of looping through each CPU core and sending an Inter-processor Interrupt (IPI). As we pointed out earlier, the TLB invalidation IPI loop becomes extremely expensive when frequent page faults occur simultaneously on a large number of CPU cores [14].

However, as we will show later in Section 5.2, in many HPC applications the computation area (the memory area on which computation takes place) is divided among CPU cores and only a relatively small part of the memory is utilized for communication. Consequently, CPU cores do not actually access the entire computation area and when an address mapping is modified most of the CPU cores are not affected. However, the information of which cores' TLB have to be invalidated is not available due to the centralized book-keeping of address translations in the address space wise page tables.

In order to overcome this problem we have already proposed per-core *partially separated page tables (PSPT)*, which is shown in Figure 3. In PSPT each core has its own last level page table, i.e., Page Global Directory (PGD). Kernel-space and regular user-space mappings point to the same Page Middle Directories (PMD), and thus, use the same PTEs to define the address space (regular boxes in the top of Figure 3). However, for the computation area per-core private page tables are used (denoted by dashed boxes in Figure 3). There are multiple benefits of such arrangement. First, each CPU core sets up PTEs exclusively for addresses

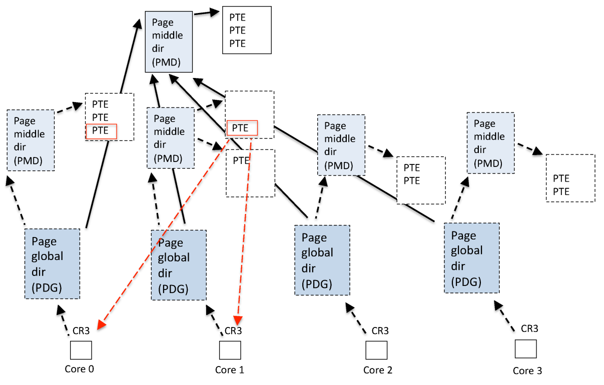


Figure 3: Per-core Partially Separated Page Tables.

that it actually accesses. Second, when a virtual to physical mapping is changed, it can be precisely determined which cores' TLB might be affected, because only the ones which have a valid PTE for the particular address may have cached a translation. Consider the red dashed lines in Figure 3, PTE invalidation in case of regular page tables require sending an IPI for each core, while PSPT invalidates the TLB only on $Core_0$ and $Core_1$. Third, synchronization (particularly, holding the proper locks for page table modifications) is performed only between affected cores, eliminating coarse grained, address space wise locks that are often utilized in traditional operating system kernels [9].

It is also worth pointing out, that the private fashion of PTEs does not imply that mappings are different, namely, private PTEs for the same virtual address on different cores define the same virtual to physical translation. When a page fault occurs, the faulting core first consults other CPU cores' page tables and copies a PTE if there is any valid mapping for the given address. Also, when a virtual address is unmapped, all CPU cores' page table, which map the address, need to be modified accordingly. This requires careful synchronization during page table updates, but the price of such activity is much less than constant address space wise TLB invalidations. For further discussion on PSPT refer to [14].

Note that an alternative solution to the careful software approach could be if the hardware provided the right capability to invalidate TLBs on multiple CPU cores, such as special instructions for altering TLB contents on a set of CPU cores. Thus, although we do provide an OS level solution in this paper, we would encourage hardware vendors to put a stronger focus on TLB invalidation methods for many-core CPUs.

3. CORE-MAP COUNT BASED PAGE REPLACEMENT

State of the art page replacement algorithms, such as approximations of the LRU policy, aim at minimizing the number of page faults during execution by means of estimating the working set of the application and keeping it in RAM [11]. To estimate the working set, the operating system monitors memory references, relying on the access bit of page table entries. Specifically, the OS scans PTEs checking and clearing the accessed bit in a periodic fashion. In case of the x86 architecture, however, every time the accessed bit is cleared, it is also required to invalidate the TLB entry (on all affected CPU cores) corresponding to the given PTE in order to ensure that the hardware will set the access bit when the page is referenced again.

We have already stated above that the many-core co-processor accessing the host memory setup is significantly different than the multi-core CPU based host machine accessing a regular disk configuration. First, the PCI Express bus is orders of magnitude faster (we measured up to 6GB/s bandwidth between the host and the MIC) than accessing a regular disk, which makes the relative price of data transfer during page fault handling less expensive compared to the disk based scenario. Second, as we pointed out previously, the large number of CPU cores on the co-processor renders the price of remote TLB invalidations much higher, when frequent page faults occur simultaneously in multiple threads [14].

As we will demonstrate through quantitative measurements in Section 5, LRU can successfully decrease the number of page faults (compared to the basic FIFO policy), but the price of frequent TLB invalidations for obtaining page usage statistics outweighs the benefits of the algorithm itself, rendering the performance of LRU lower than FIFO. We emphasize that the problem with LRU on manycore CPUs is not the original policy how victim pages are chosen. It is the *overhead of collecting information* so that the policy can be realized.

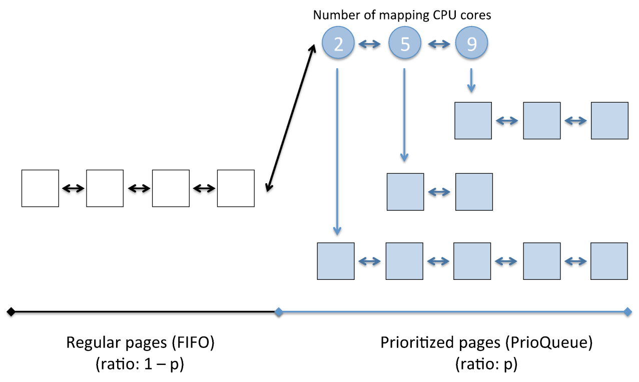


Figure 4: Core-Map Count based Priority Replacement. ***The ratio of prioritized pages is defined by*** p***, where*** $0 <= p <= 1$.

In order to overcome the aforementioned issue, we propose a novel page replacement policy that exploits the auxiliary knowledge of the number of mapping CPU cores of each memory page, which we gather from the per-core partially separated page tables. Note that such information cannot be obtained from regular page tables due to their centralized book keeping of address mappings. As an example, in Figure 3 one can see that the page corresponding to the red circled PTE is mapped by two CPU cores. Intuitively, pages that are mapped by a large number of CPU cores (and thus have been accessed by them) are likely more important than per-core local data. Furthermore, remapping a page that has been referenced by a large number threads requires TLB invalidations on all the mapping CPU cores, while pages that are per-core private (or mapped by only a few cores) imply less time spent on TLB invalidation.

The high level design of the algorithm, which we call *Core Map Count based Priority replacement* (CMCP), is shown in Figure 4. Physical pages are separated into two groups, regular pages (left side of the Figure) are maintained on a simple FIFO list, while the priority pages (shown on the right) are held on a priority queue according to the number of mapping CPU cores for each page. The parameter p of the algorithm defines the ratio of prioritized pages. With p converging to 0, the algorithm falls back to the simple FIFO replacement, while p approaching 1, all pages are ordered by the number of mapping CPU cores. The motivation behind introducing p is to allow an optimization method to discover the appropriate ratio which yields the best performance. In Section 5, we will provide measurements on the effect of varying the ratio.

When a new PTE is set up by a CPU core, it first consults PSPT to retrieve the number of mapping cores for the particular page, and it tries to place the page into the prioritized group. If the ratio of prioritized pages already exceeds p and the number of mapping cores of the new page is larger than that for the lowest priority page in the prioritized group, then the lowest priority page is moved to FIFO and the new page is placed into the priority group. Otherwise, the new page goes to the regular FIFO list. Gradually, pages with the largest number of mapping CPU cores end up in the priority group. In order to enable moving pages to the other direction (i.e., from the prioritized group to the FIFO group), we employ a simple aging method, where all prioritized pages slowly fall back to FIFO. Such mechanism is required so that prioritized pages which are not used any more can also be swapped out, and thus preventing the priority group from being monopolized by such pages.

As for eviction, the algorithm either takes the first page of the regular FIFO list, or if the regular list is empty, the lowest priority page from the prioritized group is removed.

The most important thing to notice is that there are no extra remote TLB invalidations involved in the decision process. Although one could intentionally construct memory access patterns for which this heuristic wouldn't work well, as we will show in Section 5, CMCP consistently outperforms both FIFO and LRU on all applications we have evaluated. It is also worth mentioning, that although we use LRU as the basis of comparison, other algorithms such as the least frequently used (LFU) policy or the clock algorithm also rely on the access bit of the PTEs [6], and thus would suffer from the same issues of extra TLB invalidations.

4. XEON PHI 64KB PAGE SUPPORT

This Section describes the 64kB page support of the Xeon Phi. The MIC features 4kB, 64kB, and 2MB page sizes, although the 64kB support currently goes unused in the Intel modified Linux kernel [17].

The 64kB page extension was originally added to create an intermediate step between 4kB and 2MB pages, so that reducing TLB misses and preserving high granularity can be attained at the same time. Figure 5 illustrates the page table format. Support can be enabled for 64kB pages via a hint bit addition in page table entries for which the hardware mechanism relies upon the operating system manipulating the page tables and address maps correctly. As indicated by $PageFrame_k$ in Figure 5, to set a 64kB mapping, the operating system must initialize 16 regular 4kB page table entries (PTE), which are a series of subsequent 4kB pages of a contiguous 64kB memory region. Furthermore, the first entry of the 16 entries must correspond to a 64kB aligned virtual address, which in turn must map to a 64kB aligned physical frame. The OS then sets a special bit of the PTE to indicate that CPU cores should cache the PTE using a 64kB entry rather than a series of separate 4kB entries, denoted by the flag 64 on the header of the PTEs in Figure 5.

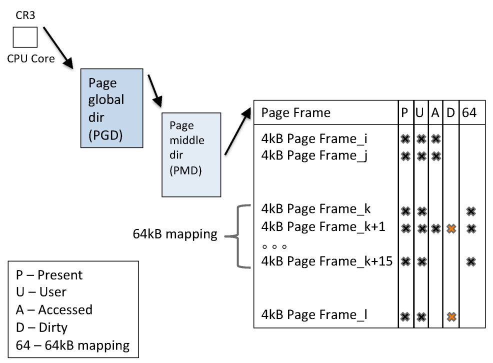


Figure 5: Xeon Phi 64kB page table entry format.

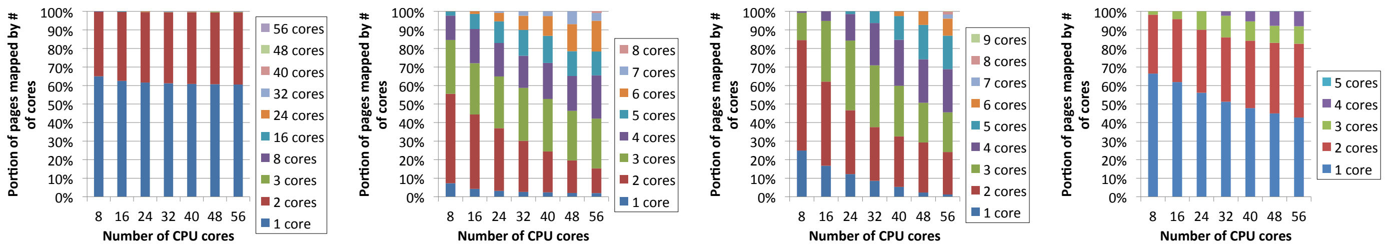
On a TLB miss, the hardware performs the page table walk as usual, and the INVLPG instruction also works as expected. On the contrary, page attributes set by the hardware work in a rather unusual way. For instance, upon the first write instruction in a 64kB mapping the CPU sets the dirty bit of the corresponding 4kB entry (instead of setting it in the first mapping of the subsequent 16 mappings as one might expect). This is indicated by the dirty bit set only for $PageFrame_{k+1}$. The accessed bit works similarly. In consequence, the operating system needs to iterate the 4kB mappings when retrieving statistical information on a 64kB page. One of the advantages of this approach is that there are no restrictions for mixing the page sizes (4kB, 64kB, 2MB) within a single address block (2MB). With respect to OS level hierarchical memory management, 64kB pages also come with multiple benefits. On one hand, they offer lower TLB miss rate compared to 4kB pages, while on the other hand, they allow finer grained memory management than using 2MB large pages. We will provide quantitative results on using 64kB pages below.

5. EVALUATION

5.1 Experimental Setup and Workloads

Throughout our experiments the host machine was an Intel$^{®}$ Xeon$^{®}$ CPU E5-2670, with 64 Gigabytes of RAM. For the manycore co-processor we used the *Knights Corner* Xeon Phi 5110P card, which is connected to the host machine via the PCI Express bus. It provides 8GB of RAM and a single chip with 60 1.053GHz x86 cores, each processor core supporting a multithreading depth of four. The chip includes coherent L1 and L2 caches and the inter-processor network is a bidirectional ring [15].

We use the OpenMP [24] version of three representative algorithms from the *NAS Parallel Benchmarks* [2]. Namely, CG (Conjugate Gradient), LU (Lower-Upper symmetric Gauss-Seidel) and BT (Block Tridiagonal). We chose not to include the other benchmarks from the NAS Parallel set for the following reasons. EP (Embarrassingly Parallel) uses very



(a) cg.B (b) lu.B (c) bt.B (d) SCALE (sml)

Figure 6: Distribution of pages according to the number of CPU cores mapping them for NPB B class benchmarks and SCALE (512MB).

small amount of memory and thus hierarchical memory management is not necessary. It has been already pointed out by [27] that FT (Fourier-Transformation) and MG (three-dimensional discrete Poisson equation) are highly memory intensive workloads. We found that without algorithmic modifications, such as shown in [30], running these applications in an out-of-core fashion is not feasible. Finally, IS (Integer Sort) doesn't appear to have high importance for our study.

However, we use the *Scalable Computing for Advanced Library and Environment* (SCALE) [3], a stencil computation code for weather and climate modelling developed at RIKEN AICS. SCALE is a complex stencil computation application, which operates on multiple data grids. It is written in Fortran 90 and it also uses OpenMP to exploit thread level parallelism.

For each benchmark we used two configurations with respect to memory usage. Small configuration, i.e., B class NPB benchmarks and 512 megabytes memory requirement for SCALE, which we denote by SCALE (sml) were used for experiments using only 4kB pages, while C class NPB benchmarks and a 1.2GB setup of SCALE, denoted by SCALE (big) were utilized for the comparison on the impact of different page sizes. For all applications we used Intel's compiler with the $-O3$ flag and verified in the compiler log that special vector operations for the Xeon Phi^{TM} were indeed generated. In order to move the computation data into the PSPT memory region we interface a C block with the Fortran code which explicitly memory maps allocations to the desired area. It is also worth mentioning that in all experiments we mapped application threads to separate CPU cores, partially due to the fact that we dedicated some of the hyperthreads to the page usage statistics collection mechanism for LRU.

Regarding the discussion on page replacement policies, we will be comparing our proposal against an LRU approximation, which implements the same algorithm employed by the Linux kernel [22]. It tracks pages on two lists, the *active* and *inactive* queues, where *active* denotes pages which are derived as part of the application's working set, while *inactive* represents the rest of the memory. Pages transit between these two states based on periodic scanning of the access bit of the corresponding PTEs, which is carried out in a timer set for every 10 milliseconds. As briefly mentioned above, we use dedicated hyperthreads for collecting page usage statistics, i.e., the timer interrupts are not delivered to application cores so that interference with the application is minimized.

5.2 Page Sharing among CPU Cores

As we mentioned above, we found that in various HPC applications the footprint of CPU cores accessing pages on the computation area is surprisingly regular. Figure 6 illustrates the result of our analysis, which we obtain from the per-core page tables of PSPT. For each application we show the distribution of pages (in the computation area) according to the number of CPU cores that access them. The key observation is that for all applications listed, regardless the overall number of CPU cores involved, the majority of pages are shared by only a very few cores.

Specifically, in case of both CG and SCALE (stencil computation) over 50% of the pages are core private. Furthermore the remaining pages are mainly shared by only two cores. LU and BT show somewhat less regular pattern, nevertheless, the majority of pages are still mapped by only less than six cores and over half of them are mapped by at most three. This implies that with PSPT, every time when a page is swapped out only a few CPU cores' TLB need to be flushed as opposed to the every CPU scenario that regular page tables require.

5.3 Relative Performance to Memory Constraint

Before moving on to demonstrate scalability of PSPT, as well as the impact of various page replacement policies, we provide measurements on how the degree of memory constraint imposed on various applications affects performance.

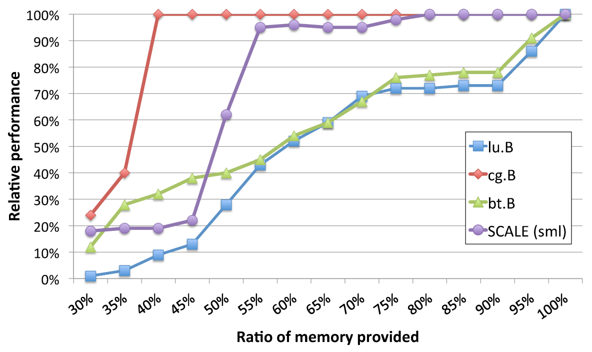
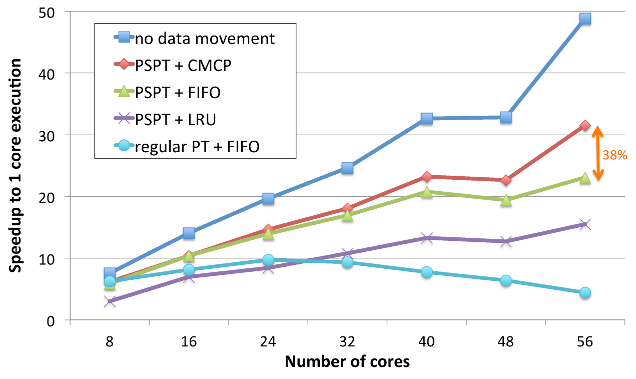
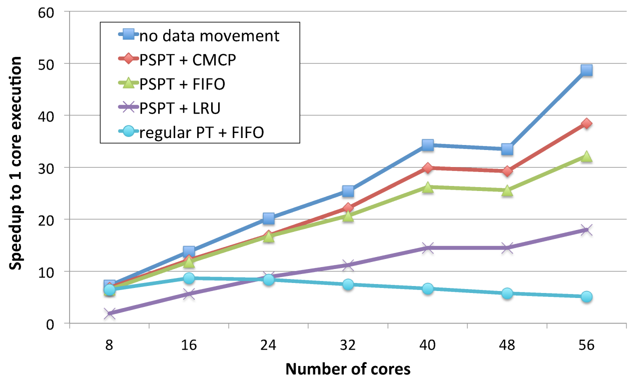


Figure 8: Relative performance with respect to physical memory provided for NPB B class benchmarks and SCALE.

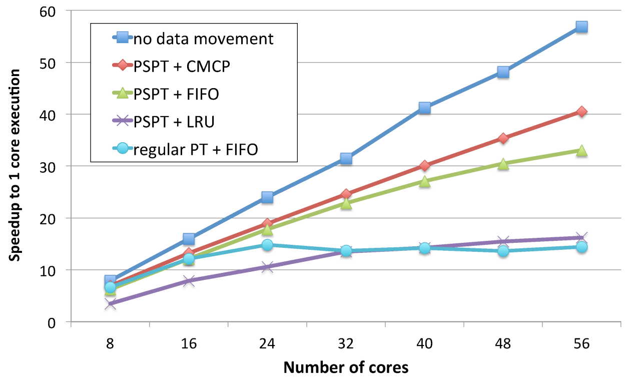
Figure 8 illustrates the measurements for small size benchmarks using PSPT with 4kB pages, 56 CPU cores, and FIFO



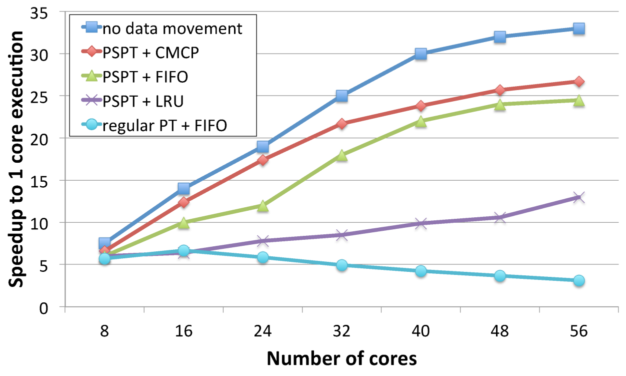
(a) bt.B



(b) lu.B



(c) cg.B



(d) SCALE (sml)

Figure 7: Performance measurements of NAS Parallel benchmarks and SCALE (sml) comparing regular page tables and PSPT using various page replacement policies.

replacement policy. The X axis shows the ratio of memory provided and Y axis represents the relative performance compared to the case where no data movement takes place. As seen, two kinds of behavior can be distinguished. LU and BT shows gradual decrease in performance with respect to the amount of memory provided, which is immediately visible once the physical memory available is less than 100% of the application's requirement.

On the other hand, CG and SCALE doesn't suffer significant performance degradation until approximately 35% and 55%, respectively. We believe this is due to sparse representation of data in these applications. Nevertheless, once the turning point is crossed, performance starts dropping steadily for both of the benchmarks.

In order to stress our kernel's virtual memory subsystem, in the rest of the experiments we set the memory constraint so that relative performance with FIFO replacement results between 50% and 60% for each application.

5.4 PSPT and Page Replacement Policies

We will now provide runtime measurements for the three benchmarks we evaluated from the NAS parallel suite and for RIKEN's SCALE. We are focusing on the benefits of partially separated page tables with the combination of various page replacements policies, as well as on their scalability with the number CPU cores. As it has been discussed before (and shown in Figure 6), all of these applications have very regular page sharing pattern among the CPU cores, where a large fraction of the pages are mapped by only a few CPU cores, suggesting significant benefits using PSPT. Results are shown in Figure 7.

For each benchmark we ran five configurations. First, using regular page tables and providing sufficient physical memory so that data movement does not occur. This is indicated by the legend *no data movement*. As mentioned above, we limit physical memory so that FIFO replacement using PSPT achieves approximately half of the performance of the *no data movement* configuration. This translates to physical memory limitation of 64% for BT, 66% for LU, and 37% in case of CG and approximately half of the memory requirement of SCALE. We measured the performance of FIFO replacement for both regular and partially separated page tables, indicated by *regular PT + FIFO* and *PSPT + FIFO*, respectively. Additionally, we compare the performance of page replacement policies by evaluating the effect of LRU, denoted by *PSPT + LRU*, and Core-Map Count Based replacement, indicated by *PSPT + CMCP*.

The first thing to point out is the fact that there is nothing wrong with regular page tables in case no data movement (and thus no address remapping) is performed by the OS. However, when frequent page faults occur concurrently on several cores, regular page tables hardly scale up to 24 cores,

App.	Policy	Attribute	8 cores	16 cores	24 cores	32 cores	40 cores	48 cores	56 cores
bt.B	FIFO	page faults	2726737	1362879	912466	643175	507325	422293	374839
		remote TLB invalidations	12429404	7363887	5573193	4373695	3803838	3465130	3226009
		dTLB misses	317301578	158902560	104686518	78861340	63265002	52566943	45131883
	LRU	page faults	2081372	1121643	736023	526239	405079	343081	283699
		remote TLB invalidations	36835046	27810383	19186674	13964030	10818576	9064231	7200812
		dTLB misses	329974700	181141377	121038176	90000002	69873462	56684949	48067965
	CMCP	page faults	2095202	1166479	707894	515089	355083	316789	262958
		remote TLB invalidations	8054343	5007407	3492531	2772393	1968787	1932189	1683682
		dTLB misses	303861675	154641090	102907981	77448482	61478878	51527221	43876503
cg.B	FIFO	page faults	1201555	612630	416332	316925	257412	214623	**170332**
		remote TLB invalidations	5135525	2804708	1956586	1518976	1251353	1061759	848976
		dTLB misses	2094329816	1047163487	698202996	523651167	418949006	349143469	299118728
	LRU	page faults	697582	204362	159474	118680	103060	85709	**74393**
		remote TLB invalidations	22972046	10429447	5799486	3472687	2551439	1913195	1536147
		dTLB misses	2102332886	1051392986	700485750	524945408	419223430	348950189	298896012
	CMCP	page faults	974695	507322	355728	273488	219813	169125	**147269**
		remote TLB invalidations	4097411	2270867	1629661	1266941	1024237	788419	689344
		dTLB misses	2092475232	1046394257	697830654	523418261	418753017	348865565	299034762
lu.B	FIFO	page faults	1664098	736180	474095	352133	281916	235222	203914
		remote TLB invalidations	7670041	4370794	3182328	2575556	2166993	1953164	1703195
		dTLB misses	1198077215	599572624	400681210	301074654	241110822	201467566	173098992
	LRU	page faults	1404043	702963	469002	341749	249638	201451	201691
		remote TLB invalidations	184015529	62149332	30745581	19286652	12227868	10137298	7161364
		dTLB misses	1286055611	653339411	431973663	323255046	253339298	209722670	178214335
	CMCP	page faults	849638	534063	415147	260252	172057	153015	159575
		remote TLB invalidations	3791955	2747828	2289613	1476379	1024914	979601	1094896
		dTLB misses	1195314147	598279823	399867708	300469205	240438760	200940292	172782035
SCALE	FIFO	page faults	1689552	845232	563854	171636	116877	96999	83817
		remote TLB invalidations	6612775	3566425	2547823	817612	579835	497952	439276
		dTLB misses	153176016	77193205	52302140	39044538	31526874	26470846	22932556
	LRU	page faults	272450	145358	98365	73234	55315	45500	32988
		remote TLB invalidations	17091293	7262438	4469165	3374882	2674781	2224822	1776893
		dTLB misses	157644315	79764075	54112492	41294510	33055668	27486007	23655615
	CMCP	page faults	698334	256230	137778	92977	73260	61343	62057
		remote TLB invalidations	2522939	985460	545405	382940	312692	269734	281294
		dTLB misses	150999236	75899452	51330254	38831504	31412049	26374264	22871311

Table 1: Per CPU core average number of page faults, TLB invalidations, and TLB misses for various workloads and page replacement policies as the function of the number of CPU cores utilized by the application.

resulting in completely unacceptable performance. In fact, one can observe slowdown in most cases when more than 24 cores are utilized.

On the other hand, partially separated page tables provide relative speed-ups (i.e., scalability) similar to the no data movement configuration. Considering page replacement policies, surprisingly, we found that LRU yields lower performance than FIFO, which we will discuss in more detail below. Nevertheless, the key observation with regards to page replacement policies is the superior performance of the Core-Map Count based Replacement policy, which consistently outperforms FIFO, yielding 38%, 25%, 23%, and 13% better results when running on 56 CPU cores for BT (Figure 7a), LU (Figure 7b), CG (Figure 7c), and SCALE (Figure 7d), respectively.

5.5 What is wrong with LRU?

In order to gain a deeper understanding of LRU's behavior we logged various attributes of the execution. Table 1 summarizes the data. We provide per-core average values for the number of page faults, remote TLB invalidations (i.e., TLB invalidation requests coming from other CPU cores), and data TLB misses. As seen, LRU *successfully decreases the number of page faults* compared to FIFO for all benchmarks. We highlighted some of the values for CG to further emphasize the difference. However, it comes at the price of substantial (up to several times) increase in the number of remote TLB invalidations. Contrary to our expectations, the number of dTLB misses isn't increased significantly by LRU's page scanning mechanism, mainly because there is a large number of TLB misses anyway. Notice that TLB misses do not only stem from TLB invalidations, but simply from the fact that the size of the TLB cache is insufficient for covering the entire address range mapped by the application. Moreover, although it is not included in the table, we also observe up to 8 times increase in CPU cycles spent on synchronization (i.e., locks) for remote TLB invalidation request structures. Altogether, we confirmed that the above mentioned components account for the performance degradation seen in case of LRU. On the other hand, while CMCP also reduces the number of page faults compared to FIFO, supporting the assumption with respect to the importance of pages mapped by multiple CPU cores, it does not introduce any overhead from the above mentioned issues.

It is also worth pointing out that we did experiment with adjusting the frequency of LRU's page scanning mechanism. In principle, the lower the frequency is, the less the TLB invalidation overhead becomes. However, doing so defeats the very purpose of LRU, i.e., decreasing the number of page faults. Eventually, with very low page scanning frequency LRU simply fell back to the behavior of FIFO. Nevertheless, whenever LRU succeeded in decreasing the number of page faults, it always yielded worse performance than FIFO.

5.6 The role of p in CMCP

As we mentioned earlier in Section 3, we provide measurements on the effect of the ratio of prioritized pages for CMCP policy. Figure 9 shows the results. As seen, the ratio of prioritized pages affects performance improvement quite significantly, moreover, the impact is also very workload specific. For instance, CG benefits the most from a low ratio, while in case of LU or SCALE high ratio appears to work better. We adjusted the algorithm's parameter manually in this paper, but determining the optimal value dynamically based on runtime performance feedback (such as page fault frequency) is part of our future work.

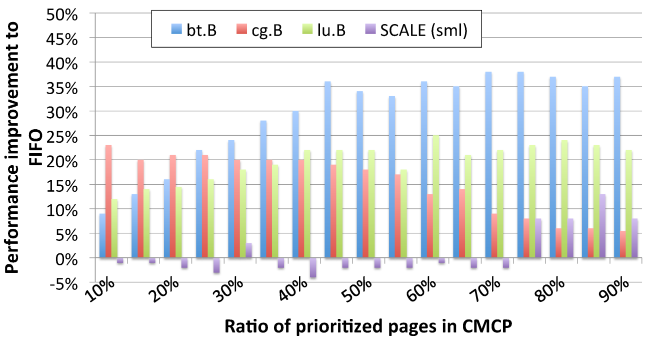


Figure 9: The impact of the ratio of prioritized pages in CMCP.

One could also argue that the number of mapping cores of a given page is dynamic with the time changing which in turn naturally impacts the optimal value of p. Although we believe the applications we consider in this paper exhibit rather static inter-core memory access patterns, a more dynamic solution with periodically rebuilding PSPT could address this issue as well.

5.7 The Impact of Different Page Sizes

In this Section we turn our attention towards the impact of various page sizes (i.e., 4kB, 64kB and 2MB) supported by the Intel® Xeon Phi™. Using larger pages reduces the number of TLB misses and thus can improve performance. However, in case of OS level data movement there are two problems when it comes to larger pages.

First, the usage of larger pages also implies that every time a page is moved to or from the host memory, significantly more data needs to be copied. Second, since with larger pages the granularity of the memory is more coarse grained, the probability of different CPU cores accessing the same page is also increased, and consequently, the price of remote TLB invalidations when the corresponding address is remapped. For example, using 2MB pages is 512x more coarse-grained than the regular 4kB pages, but on the other hand, 64kB pages yield only 16 times more coarse-grained mappings, possibly benefiting more the hierarchical memory scenario. Therefore, the main issue we seek to investigate is how the increased price of the data movement and TLB invalidations compare to the benefits of using larger pages. We are also interested in seeing how the imposed memory constraint influences performance.

We used the C class NPB benchmarks and SCALE with 1.2GB of memory. Figure 10 illustrates the results. Note that we leave the investigation of combining various page replacement policies with different page sizes as future work and we present results only for FIFO replacement in this paper. Nevertheless, for all measurements, we used 56 CPU cores.

As expected, when memory constraint is low, large pages (i.e., 2MB) provide superior performance compared to both 4kB and 64kB pages. However, as we decrease the memory provided, the price of increased data movement quickly outweighs the benefits of fewer TLB misses. Due to their finer granularity, the higher the memory constraint is the more we can benefit from smaller pages. This tendency can be clearly seen in case of BT and LU (Figure 10a and 10b, respectively), where with the decreasing memory first 64kB pages and later 4kB pages prove to be more efficient. On the other hand, we found that for CG and SCALE (Figure 10c and 10d, respectively), 64kB pages consistently outperform 4kB pages even when memory pressure is already considerably high. We believe further increasing the memory pressure would eventually produce the expected effect, but it is yet to be confirmed.

Indeed, the operating system could monitor page fault frequency and adjust page sizes dynamically so that it always provides the highest performance. At the same time, different page sizes could be used for different parts of the address space. Mapping frequently accessed areas with large pages could reduce TLB misses, while data that need to be often moved back and forth could benefit more from smaller page sizes. Exploring these directions is part of our future plans.

6. RELATED WORK

6.1 Operating Systems for Manycores

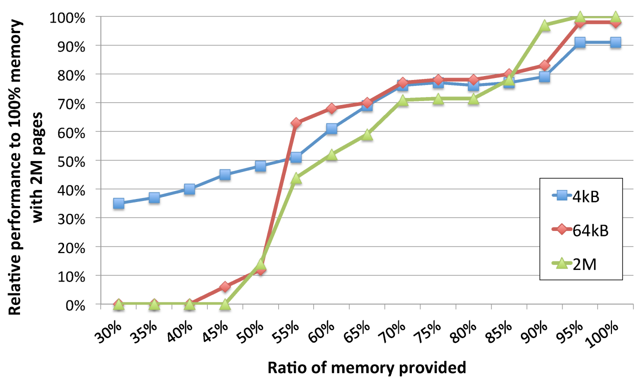
Operating system organization for manycore systems has been actively researched in recent years. In particular, issues related to scalability over multiple cores have been widely considered.

K42 [12, 20] was a research OS designed from the ground up to be scalable. It's *Clustered Object* model provides a standard way for implementing concurrently accessed objects using distribution and replication, the same principles we applied to page tables. At the time of K42, nevertheless, there were no co-processors or multiple levels of memory hierarchy.

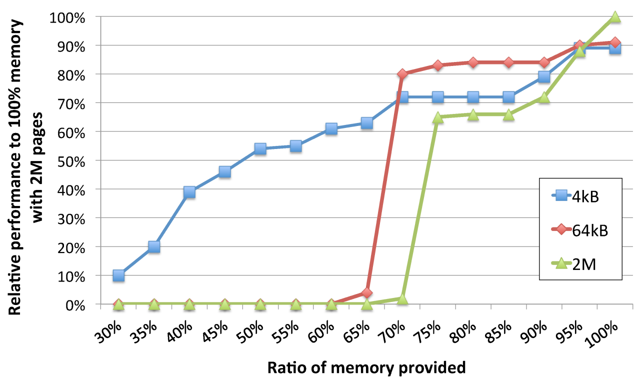
Corey [7], an OS designed for multicore CPUs, argues that applications must control sharing in order to achieve good scalability. Corey proposes several operating system abstractions that allow applications to control inter-core sharing. For example, Corey's *range* abstraction provides means to control whether a memory area is private or shared by certain CPU cores, however, in a Corey range all CPU cores share the same set of page tables, and thus the TLB problem we address cannot be handled by their solution. Moreover, we are also aiming at application transparency.

Barrelfish [28] argues that multiple types of diversity and heterogeneity in manycore computer systems need to be taken into account. It represent detailed system information in an expressive "system knowledge base" accessible to applications and OS subsystems and use this to control tasks such as scheduling and resource allocation. While we explicitly address the Intel® Xeon Phi™ product family in this paper, system knowledge base, as proposed in Barrelfish could be leveraged for placing threads to CPU cores that have low IPI communication cost so that TLB invalidations can be performed more efficiently.

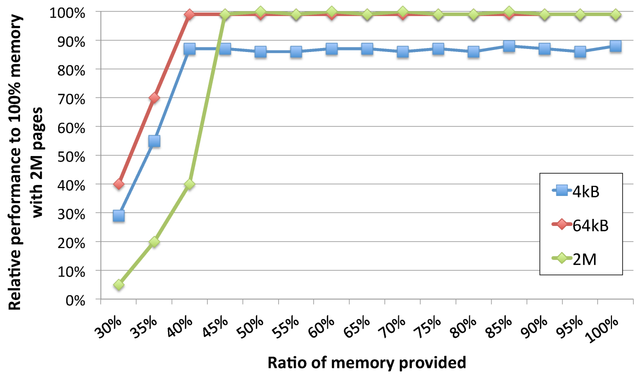
Scalable address spaces in particular have been also the focus of recent research. Clements et. al [9] proposed increasing the concurrency of kernel operations on a shared



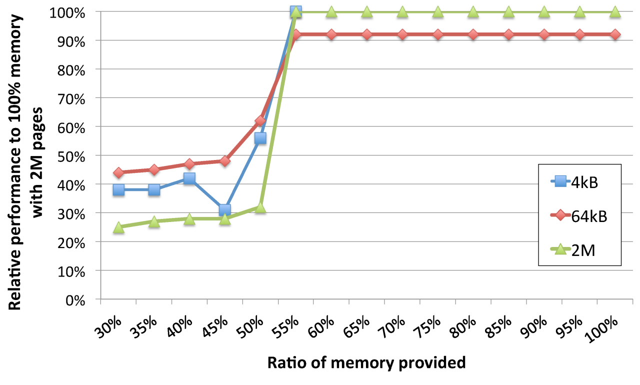
(a) bt.C



(b) lu.C



(c) cg.C



(d) SCALE (big)

Figure 10: The impact of page sizes on relative performance with respect to memory constraint for various benchmarks.

address space by exploiting read-copy-update (RCU) so that operations that mutate the same address space can avoid contention on shared cache lines. Moreover, published at the same time with our previous proposal [14], they also explored the idea of per-core page tables [10], however, neither hierarchical memory nor page replacement policies are considered in their study.

An idea, similar to PSPT, has been discussed by Almaless and Wajsburt [5]. The authors envision replicating page tables in NUMA environments to all memory clusters in order to reduce the cost of address translations (i.e., TLB misses) on CPU cores, which are located far from the otherwise centralized page tables. Although their proposal is similar to ours, they are addressing a very NUMA specific issue, furthermore, they do not provide an actual implementation.

In the context of heterogeneous kernels, IBM's FusedOS [25] also promotes the idea of utilizing different kernel code running on CPU cores dedicated to the application and the OS. However, they do not focus on hierarchical memory systems. GenerOS [32] partitions CPU cores into application core, kernel core and interrupt core, each of which is dedicated to a specified function. Again, the idea of utilizing dedicated cores for system call execution is similar to the utilization of the host machine for offloading system calls from the co-processor.

Villavieja et. al also pointed out the increasing cost of remote TLB invalidations with the number of CPU cores in chip-multiprocessors (CMP) systems [31]. In order to mitigate the problem the authors propose a lightweight hardware extension (a two-level TLB architecture that consists of a per-core TLB and a shared, inclusive, second-level TLB) to replace the OS implementation of TLB coherence transactions. While the proposed solution yields promising results, it requires hardware modifications, which limits its applicability. To the contrary, our proposal offers a solution entirely implemented in software.

6.2 Programming Models

Programming models for accelerators (i.e., co-processors) have also been the focus of research in recent years. In case of GPUs, one can spread an algorithm across both CPU and GPU using CUDA [23], OpenCL [19], or the OpenMP [24] accelerator directives. However, controlling data movement between the host and the accelerator is entirely the programmer's responsibility in these models. Nevertheless, a recent announcement by Nvidia reveals *Unified Memory*, a new feature in the upcoming CUDA 6 release [4]. Unified memory will allow the programmer to have a unified view of the host and the device memory on GPUs, eliminating the need to manually orchestrate data movement. Although Nvidia states their mechanism works on the memory page level, no details have been disclosed regarding their page replacement policy.

OpenACC [8] allows parallel programmers to provide directives to the compiler, identifying which areas of code to accelerate. Data movement between accelerator and host memories and data caching is then implicitly managed by the compiler, but as the specification states, the limited device memory size may prohibit offloading of regions of code that operate on very large amounts of data.

Intel provides several execution models for the Xeon Phi^{TM} product family [16]. One of them, the so called *Mine-Your-Ours* (MYO), also referred to as *Virtual Shared Memory*, provides similar features to Nvidia's unified memory, such as transparent shared memory between the host and the co-processor. However, at the time of writing this paper, the main limitation of MYO is that the size of the shared memory area cannot exceed the amount of the physical memory attached to the co-processor. On the contrary, we explicitly address the problem of dealing with larger data sets than the amount of physical memory available on the co-processor card.

Other memory models have been also proposed for GPUs, the *Asymmetric Distributed Shared Memory* (ADSM) maintains a shared logical memory space for CPUs to access objects in the accelerator physical memory but not vice versa. When a method is selected for accelerator execution, its associated data objects are allocated within the shared logical memory space, which is hosted in the accelerator physical memory and transparently accessible by the methods executed on CPUs [13]. While ADSM uses GPU based systems providing transparent access to objects allocated in the co-processor's memory, we are aiming at an approach of the opposite direction over Intel's MIC architecture.

7. CONCLUSION AND FUTURE WORK

Memory management is one of the major challenges when it comes to programming co-processor based heterogeneous architectures. To increase productivity, we have investigated the feasibility of an OS level, application transparent solution targeting the Intel Xeon Phi. Focusing on page replacement algorithms, one of our main findings is that state of the art approaches, such as approximations of the LRU policy, are not well suited for massive many-cores due to their associated cost of obtaining page usage statistics. We emphasize that the problem with LRU on many-core CPUs does not stem from the policy of keeping recently used pages close to the CPU. It is the price of frequently scanning page table entries, which requires a large number of extra TLB invalidations.

Building upon our previous proposal, per-core *Partially Separated Page Tables* (PSPT), in this paper, we have proposed *Core Map Count based Priority* (CMCP) page replacement policy that prioritizes pages based on the number of mapping CPU cores. The main advantage of our approach is the ability to eliminate remote TLB invalidations otherwise necessary for page usage tracking.

We have further evaluated PSPT on various real life applications and demonstrated its scalability to large number of CPU cores. Enhanced with CMCP, we have also shown that we consistently outperform existing page replacement policies by up to 38% when running on 56 cores. Additionally, for the first time, we have provided an implementation of the experimental 64kB page support of the Intel Xeon Phi and concluded that adequate page size is a function of the memory constraint and there is space for 64kB pages in the context of hierarchical memory management. Across various workloads, our system is capable of providing up to 70% of the native performance with physical memory limited to half, allowing essentially to solve two times larger problems without any need for algorithmic changes.

With respect to future architectures, such as the aforementioned Knights Landing chip, which will replace the PCI Express bus with printed circuit board (PCB) connection between memory hierarchies (rendering the bandwidth significantly higher), we expect to see further performance benefits of our solution. In the future, we intend to dynamically adjust page sizes during runtime in response to memory constraint as well as to integrate such decisions with page replacement policies.

Acknowledgment

This work has been partially supported by the CREST project of the Japan Science and Technology Agency (JST) and by the National Project of MEXT called Feasibility Study on Advanced and Efficient Latency Core Architecture.

We would also like to express our gratitude to Intel Japan for providing early access to the pre-production version of the hardware, as well as for the technical support associated with the Intel$^{®}$ Xeon Phi^{TM} product family.

8. REFERENCES

[1] Intel unveils 72-core x86 Knights Landing CPU for exascale supercomputing. `http://www.extremetech.com/extreme/171678-intel-unveils-72-core-x86-knights-landing-cpu-for-exascale-supercomputing`.

[2] NASA. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Software/NPB`.

[3] RIKEN AICS. Scalable Computing for Advanced Library and Environment. `http://scale.aics.riken.jp/`.

[4] Unified Memory in CUDA 6. `http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6`.

[5] Almaless, G., and Wajsburt, F. Does shared-memory, highly multi-threaded, single-application scale on many-cores? In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism* (2012), HotPar '12.

[6] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces.* Arpaci-Dusseau, 2013.

[7] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), OSDI'08, pp. 43–57.

[8] CAPS Enterprise and CRAY Inc and The Portland Group Inc and NVIDIA. The OpenACC Application Programming Interface. Specification, 2011.

[9] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scalable address spaces using RCU balanced trees. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS '12.

[10] Clements, A. T., Kaashoek, M. F., and Zeldovich, N. RadixVM: scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13.

[11] Denning, P. J. Virtual Memory. *ACM Computing Surveys 2* (1970), 153–189.

[12] Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the third symposium on Operating systems design and implementation* (1999), OSDI '99.

[13] Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N., and Hwu, W.-m. W. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (New York, NY, USA, 2010), ASPLOS '10, ACM, pp. 347–358.

[14] Gerofi, B., Shimada, A., Hori, A., and Ishikawa, Y. Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on* (may 2013).

[15] Intel Corporation. Intel Xeon Phi Coprocessor Software Developers Guide, 2012.

[16] Intel Corporation. Knights Corner: Open Source Software Stack, 2012.

[17] Jeffers, J., and Reinders, J. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann, 2013.

[18] Keckler, S., Dally, W., Khailany, B., Garland, M., and Glasco, D. GPUs and the Future of Parallel Computing. *Micro, IEEE 31*, 5 (2011), 7–17.

[19] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[20] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V. K42: building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (2006), EuroSys '06.

[21] Matsuo, Y., Shimosawa, T., and Ishikawa, Y. A File I/O System for Many-core Based Clusters. In *ROSS'12: Runtime and Operating Systems for Supercomputers* (2012).

[22] Mauerer, W. *Professional Linux Kernel Architecture.* Wrox Press Ltd., Birmingham, UK, UK, 2008.

[23] NVIDIA Corp. NVIDIA CUDA Programming Guide 2.2, 2009.

[24] OpenMP Architecture Review Board. OpenMP Application Program Interface. Specification, 2008.

[25] Park, Y., Van Hensbergen, E., Hillenbrand, M., Inglett, T., Rosenburg, B., Ryu, K. D., and Wisniewski, R. FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on* (2012), pp. 211–218.

[26] Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., and Mendelson, A. Programming model for a heterogeneous x86 platform. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 431–440.

[27] Saini, S., Chang, J., Hood, R., and Jin, H. A Scalability Study of Columbia using the NAS Parallel Benchmarks. Technical report, NASA Advanced Supercomputing Division, 2006.

[28] SchÃijpbach, A., Peter, S., Baumann, A., Roscoe, T., Barham, P., Harris, T., and Isaacs, R. Embracing diversity in the Barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems* (2008).

[29] Si, M., and Ishikawa, Y. Design of Direct Communication Facility for Many-Core based Accelerators. In *CASS'12: The 2nd Workshop on Communication Architecture for Scalable Systems* (2012).

[30] Sivan Toledo. A Survey of Out-of-Core Algorithms in Numerical Linear Algebra, 1999.

[31] Villavieja, C., Karakostas, V., Vilanova, L., Etsion, Y., Ramirez, A., Mendelson, A., Navarro, N., Cristal, A., and Unsal, O. S. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2011), PACT '11, IEEE Computer Society, pp. 340–349.

[32] Yuan, Q., Zhao, J., Chen, M., and Sun, N. GenerOS: An asymmetric operating system kernel for multi-core systems. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (april 2010), pp. 1 –10.