Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures

Balazs Gerofi*, Akio Shimada*, Atsushi Hori* and Yutaka Ishikawa*†
* *RIKEN Advanced Institute for Computational Science*
Kobe, JAPAN
† *Graduate School of Information Science and Technology*
The University of Tokyo
Tokyo, JAPAN
bgerofi@riken.jp, a-shimada@riken.jp, ahori@riken.jp, ishikawa@is.s.u-tokyo.ac.jp

***Abstract*—Heterogeneous architectures, where a multicore processor is accompanied with a large number of simpler, but more power-efficient CPU cores optimized for parallel workloads, are receiving a lot of attention recently. At present, these co-processors, such as the Intel® Xeon Phi™ product family, come with limited on-board memory, which requires partitioning computational problems manually into pieces that can fit into the device's RAM, as well as efficiently overlapping computation and communication. In this paper we propose an application transparent, operating system (OS) assisted hierarchical memory management system, where the OS orchestrates data movement between the host and the device and updates the process virtual memory address space accordingly. We identify the main scalability issues of frequent address space changes, such as the increasing price of TLB invalidations with the growing number of CPU cores, and propose *partially separated page tables* with *address-range CPU masks* to overcome the problem. With partially separated page tables each core maintains its own set of mappings of the computation area, enabling the OS to perform address space updates in a scalable manner, and involve a particular CPU core in TLB invalidation only if it is absolutely necessary. Furthermore, we propose *dedicated data movement cores* in order to efficiently overlap computation and communication. We provide experimental results on stencil computation, a common HPC kernel, and show that OS assisted memory management has the potential for scalable transparent data movement.**

I. Introduction

Although Moore's Law continues to drive the number of transistors per square mm, reducing voltage in proportion to transistor size, so that the energy per operation would be dropping fast enough to compensate for the increased density, is no longer feasible. As a result of such transition, heterogeneous architectures are becoming widespread. In a heterogeneous configuration, multicore processors, which implement a handful of complex cores that are optimized for fast single-thread performance, are accompanied with a large number of simpler, and slower, but much more power-efficient cores that are optimized for throughput-oriented parallel workloads [1].

The Intel® Xeon Phi™ product family is Intel's latest design targeted for processing highly parallel workloads. The Pre-Production Intel® Xeon Phi™ card, used in this paper, provides a single chip with a large number of x86 cores[1], with each processor core supporting a multithreading depth of four. The chip also includes coherent L1 and L2 caches and the inter-processor network is a bidirectional ring [2]. Currently, the Intel® Xeon Phi™ is implemented on a PCI card, and has its own on-board memory, connected to the host memory through PCI DMA operations. This architecture is shown in Figure 1.

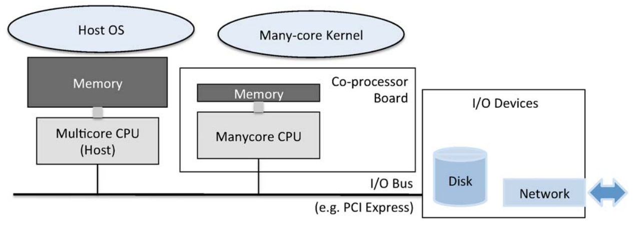


Fig. 1: **Architectural overview of a multicore host computer equipped with a manycore co-processor, connected through PCI Express.**

The on-board memory is faster than the one in the host, but it is significantly smaller. Only a few Gigabytes on the card, as opposed to the 64 GBs residing in the host machine. This limited on-board memory requires partitioning computational problems into pieces that can fit into device's RAM. At this time, it is the programmer's responsibility to partition larger computational problems into smaller pieces that can run on a co-processor and achieve high performance by efficiently overlapping computation and communication.

However, the Intel® Xeon Phi™ co-processor features a standard memory management unit (MMU), which can be utilized to provide much larger amount of virtual memory than

[1]At the time of writing this paper, the exact number of CPU cores on the Pre-Production Intel® Xeon Phi™ is confidential and the authors are under non-disclosure agreement.

978-0-7695-4996-5/13 $26.00 © 2013 IEEE
DOI 10.1109/CCGrid.2013.59

that is physically available. Just like on the multicore host, the operating system keeps track of the physical memory and manages the mapping from virtual to physical addresses. Thus, the OS running on the manycore unit can transparently move data between the card and the host, similarly how swapping is performed in a traditional OS.

Nevertheless, the scenario of manycore co-processor based memory management is considerably different than regular disk based swapping. On one hand, accessing the host memory from the co-processor is substantially faster than accessing a disk on the host, which notably reduces the cost of data movement itself. On the other hand, the large number of cores in the co-processor introduce scalability issues in the regular process address space model in case the virtual to physical mappings are frequently updated. In modern CPU architectures, which come with a Translation Lookaside Buffer (TLB) to cache memory translations, when a virtual address is remapped to a new physical page, the corresponding TLB entry has to be invalidated on each CPU core that might cache the previous mapping. Because in the regular process address space model [3], such as in Linux [4], the same set of page tables are used on all cores, the invalidation operation has to be performed on every CPU cores in the given address space. However, frequent remappings happening concurrently on all cores render the cost of TLB invalidation extremely high. Notice, however, that in many of the HPC applications (such as in stencil computation kernels), CPU cores operate mostly on their private area of the data grid and share only a relatively small fraction for communication.

Driven by this observation, we propose *partially separated page tables (PSPT)* to address the TLB problem. With PSPT each thread, thus CPU core, maintains its own set of page tables of the computation area, i.e., the memory area that is modified during computation, filling in a virtual to physical mapping only if the given core operates on the particular address. This enables the OS to perform address space updates only on affected cores when remapping a virtual address to a different physical page, and involve a particular CPU core in TLB invalidation only if it is absolutely necessary. To be able to find affected CPUs efficiently, we accompany PSPT with *address-range CPU bitmask*, which are updated consistently with the per-core page tables.

Moreover, because there is a large number of cores available on the co-processor board, some may be utilized for special purposes, such as control of data movement. We propose *dedicated data movement cores* that perform pre-fetch operations from the host memory so that computation and communication can be efficiently overlapped. We provide preliminary results on a standard stencil computation kernel and show that OS assisted memory management is capable of transparent data movement in a scalable manner. For instance, providing twice as much of virtual memory than that is available physically comes with an average of less than 9% runtime overhead when scaling up to 128 application threads.

The rest of this paper is organized as follows, Section II provides background information, Section III discusses the design of hierarchical memory management, including partially separated page tables. Section IV provides experimental results, Section V surveys related work, and finally, Section VI concludes the paper.

II. Background

RIKEN Advanced Institute of Computational Science and the Information Technology Center at the University of Tokyo have been designing and developing a new scalable system software stack for a new heterogeneous supercomputer consisting of server-grade host machines equipped with manycore co-processors.

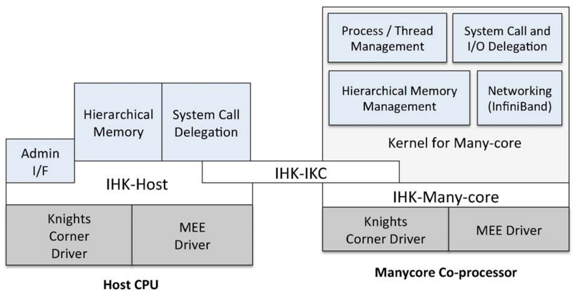


Fig. 2: **Main components of Interface for Heterogeneous Kernels (IHK) and the manycore kernel.**

Figure 2 shows the main components of the proposed software stack. The Interface for Heterogeneous Kernels (IHK) is designed to hide hardware-specific functions and provide kernel programming interfaces to operating system developers. The IHK resides in both the host and manycore units. IHK on the host is currently implemented as a Linux device driver. The inter-kernel communication (IKC) layer performs data transfer and signal notification between the host and the manycore CPUs.

We have already explored various aspects of a co-processor based system, such as scalable communication facility with direct data transfer between the co-processors [5], possible file I/O mechanisms [6], and a new process model aiming at efficient intra-node communication [7].

We are currently developing a lightweight kernel based OS targeting manycore CPUs over the IHK, and at the same time, design considerations of a hybrid execution model over the co-processor and the host multicore are also undertaken. The minimalistic kernel is designed with taking the following properties into account:

- On board memory of the co-processor is relatively small, thus, only very necessary services are provided by the kernel.
- CPU caches are also smaller, therefore, heavy system calls are shipped to and executed on the host.

III. Design

A. Execution Model

In spite of the current architecture of manycore co-processor based systems, where the co-processor is attached to a host machine (as shown in Figure 1), presumably, in the future the focus will shift towards the co-processor itself, possibly placing the host machine more and more into the background. Moreover, projections for future exascale configurations suggest that not only the number of cores per node, but also the number of nodes per system will increase dramatically. Thus, we are designing an execution model taking such transitions into account.

In order to realize scalable communication among processes running on future systems, we believe that sharing the address space among multiple cores inside a node, i.e., running a single, multi-threaded process (think of hybrid MPI and OpenMP programming), or at most a few, is the only viable approach compared to assigning separate processes to each core. Therefore, we are focusing on shared address spaces inside a node.

Figure 3 depicts the execution model under consideration. The application will be executed in a co-operative fashion between the host and the co-processor and part of the application memory (see below for further clarification) is accessible from both sides. The programmer can indicate her preference regarding where (on the host or on the co-processor) a certain part of the code should be executed.

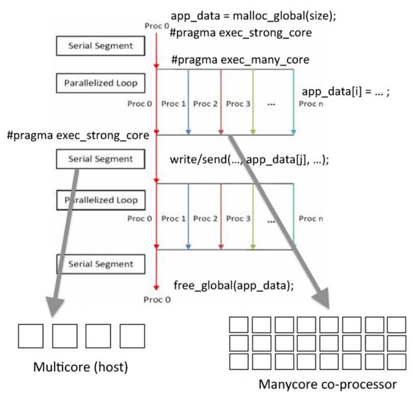


Fig. 3: **Co-processor based execution model.**

For instance, highly parallel sections of the code will be executed on the co-processor, but parts of the code that require good serial performance can be moved to the more complex cores of the host. However, due to the architecture trend mentioned above, the application is primarily executed on the co-processor and the co-processor's memory behaves akin to another level in the memory hierarchy.

Certain memory areas (which we call *global*) of the application will be transparently available on both the host and the co-processor. Such memory areas need to be allocated and freed through special library functions, *malloc_global()* and *free_global()*, respectively. The runtime system will provide these functions and it also ensures consistency for concurrent execution on the co-processor and the host CPU. As for this paper, we are concerned with the memory management system, that moves data between the host and the manycore co-processor's memory in an application transparent fashion.

B. Application Memory Layout

The application memory layout, with respect to the physical memory available on the host and the co-processor board is shown in Figure 4.

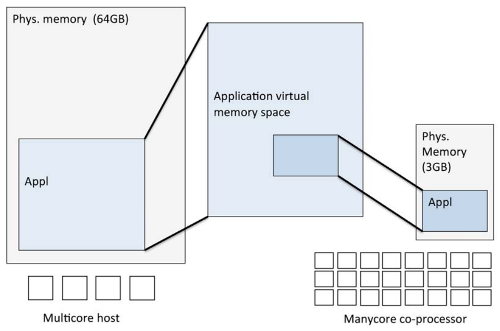


Fig. 4: **Memory layout of an application running on the manycore co-processor and the multicore host.**

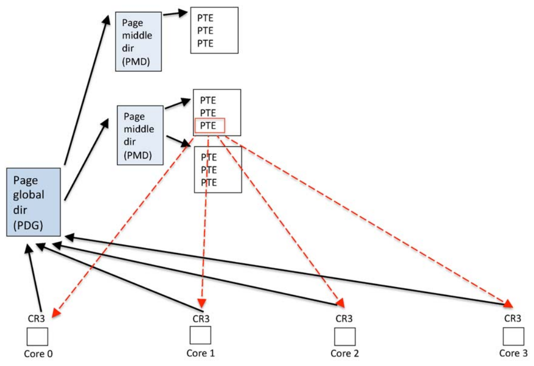
The left side of the figure illustrates the physical memory attached to the host CPU, while the right side represents the physical memory on the manycore co-processor. The application virtual address space is primarily maintained by the co-processors and as seen partially mapped onto the physical memory 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 to initiate data transfer between the host memory and the co-processor's RAM.

When the physical memory on the co-processor is almost fully utilized the kernel selects victim pages and moves the content to the host's memory. Data movement happens completely transparently from the user's point of view, essentially providing the illusion of much larger memory than the actual physical amount attached to the co-processor.

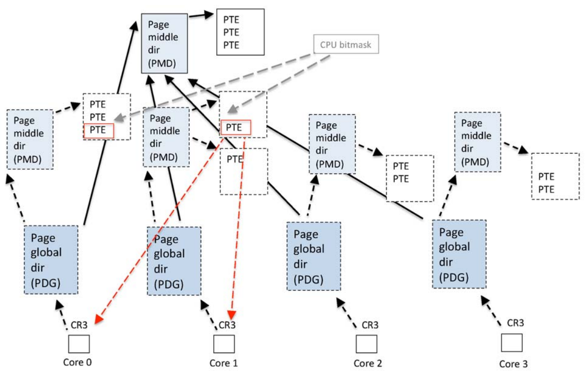
In order to retain full control over the data transfer between the co-processor and the host, we have integrated data movement directly into the virtual memory subsystem of our kernel and orchestrate data movement manually by the DMA engine residing on the co-processor. While this requires low level modifications to the operating system organization, this way we can eliminate any unnecessary software overhead both in terms of CPU consumption and additional memory footprint.

C. Partially Separated Page Tables

As it has been already mentioned above, modern CPUs cache virtual to physical mappings in the TLB. In a manycore



(a) Regular Page Tables



(b) Partially Separated Page Tables

Fig. 5: **Comparison of regular and partially separated page tables with respect to TLB invalidations.** *Dashed boxes represent per-core private page tables, and dashed red lines denote TLB invalidations.*

CPU configuration each CPU core has its own TLB and when a virtual to physical mapping is modified, it has to be ensured that all of the affected CPU cores invalidate their TLB entry for the given memory address in order to avoid using a previously cached translation.

In the regular page table configuration, where all CPU cores in an address space refer to the same set of page tables, every time a virtual to physical translation is updated all CPU cores' TLB are invalidated. In traditional operating system kernels, such as Linux, the TLB invalidation is done by means of looping through the CPU cores and sending an Inter-processor Interrupt (IPI). This configuration is shown in Figure 5a, where red dashed lines indicate IPIs.

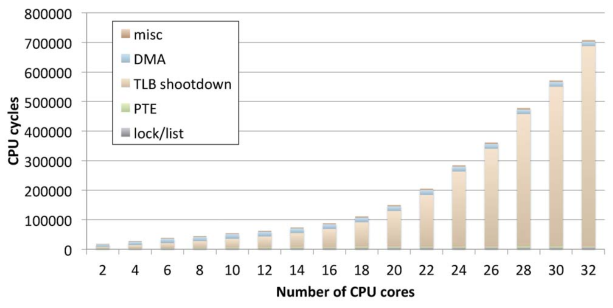
We have identified that with a large number of cores concurrently causing frequent TLB invalidations such as in a scenario where the OS often modifies virtual to physical mappings, the IPI loop becomes extremely expensive. Note, that on certain multicore architectures multicast IPI is supported by the hardware, however, the Pre-Production Intel® Xeon Phi™ does not provide with such capability.

As mentioned earlier, in case of many HPC applications, such as stencil computation kernels, the computation area is usually 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.

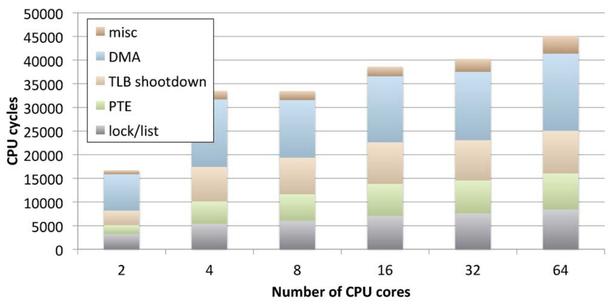
In order to overcome this problem we propose *partially separated page tables (PSPT)*, which is shown in Figure 5b. In PSPT each core has its own last level page table, i.e., Page Global Directory (PGD). Kernel-space and regular user-space mappings point to the same Page Middle Directories (PMD), and thus, use the same PTEs to define the address space (regular boxes in the top of Figure 5b). However, for the computation area per-core private page tables are used (denoted by dashed boxes in Figure 5b). There are multiple benefits of such arrangement. First, each CPU core sets up PTEs exclusively for addresses 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 5b as opposed to Figure 5a for regular page tables. As shown, PTE invalidation in case of regular page tables require sending an IPI for each core (Figure 5a), while PSPT invalidates the TLB only on Core$_0$ and Core$_1$ (Figure 5b). 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 [8].

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.

Notice, that referring other cores' page tables requires a linear search over all CPUs in the given address space. In order to speed up this search we maintain CPU bitmasks for address ranges. When a core sets a translation for a particular virtual address in its page tables, it also updates the corresponding bitmask. During TLB shootdown, or PTE modifications, only page tables for which the corresponding core is set in the given address' cpumask are iterated. This is shown in Figure 5b as *CPU bitmask*, written in grey.



(a) Regular Page Tables



(b) Partially Separated Page Tables

Fig. 6: **Cost breakdown of concurrent address remappings (page faults) with regular and partially separated page tables.** *Application is 2D heat diffusion stencil computation, data movement is performed in a synchronous fashion in the page fault handler, computation area is 2GB and the physical memory on the co-processor is limited to 1GB.*

D. Dedicated Data Movement Cores

Various recent studies have argued over the benefits of dedicating CPU cores for specific operating system tasks [9], [10]. In a dynamically changing address space, where data is constantly moved back and forth between the host and the co-processor's memory, one of the important design goals is to reduce page faults on application cores as much as possible, i.e., to transparently overlap computation and data transfer.

Because the application cores are busy performing computation, we dedicate spare CPU cores for controlling data movement. In our current prototype implementation, synchronous page faults initiate pre-fetch requests, which are then executed by data movement cores. This involves moving data in/out from/to the host, setting/unsetting mappings in the affected page tables, and requesting TLB invalidations for the corresponding CPU cores. In Section IV we present experimental results on the effect of pre-fetcher cores with respect to the reduced number of page faults.

IV. Evaluation

A. Experimental Setup

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 Pre-Production Intel® Xeon Phi™ card, which is connected to the host machine via the PCI Express bus. It provides 3GB of RAM and a single chip with a large number of x86 cores[2], 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 [2].

We provide preliminary results on our hierarchical memory system running 25 iterations of a 2D 9-point heat diffusion stencil computation kernel. We used Intel's compiler to get the best optimizations for the Xeon Phi™. Threads (i.e., CPU cores) divide the computation area among each other and only the borders are used for exchanging information. It is worth mentioning that thread assignment is static in our system, i.e., we do not migrate threads among CPU cores. Also, throughout this paper we use 4kB physical pages.

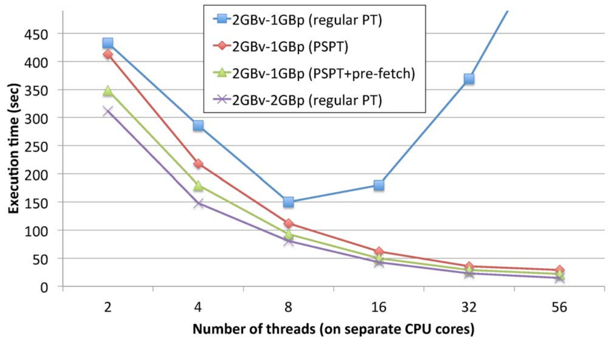
B. Page faults and TLB invalidation

The first thing we measured is the detailed cost breakdown of concurrent page faults with respect to the number of CPU cores involved when executing the heat diffusion stencil computation. The computation area of the application was 2GB and the physical memory was limited to 1GB. Data movement was performed in a synchronous fashion inside the page fault handler, using FIFO page replacement policy. Figure 6 shows the results.

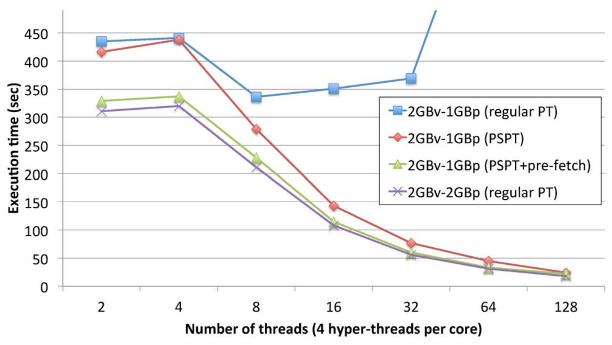
The indicated components for page faults are: managing paging data structures (*lock/list*); modifying page table entries (*PTE*); remote TLB invalidation (*TLB shootdown*); data transfer (*DMA*); and other miscellaneous operations (*misc*). Figure 6a shows the results for regular page tables, while Figure 6b depicts the measurements when partially separated page tables are used. Notice the difference between the scale of the core numbers. The most important observation is that with regular page tables the price of TLB shootdown grows faster than linearly, which has a visible impact on runtime as soon as over 16 cores. (Section IV-C provides runtime measurements.) On one hand, each CPU core spends an increasing amount of time interrupting all other cores in the address space, while on the other hand, each core receives an increasing amount of simultaneous TLB invalidation requests.

On the contrary, partially separated page tables enable the kernel to involve only the affected CPU cores in TLB invalidations, which yields much better scalability for the stencil computation scenario. As Figure 6b shows the cost of page faults stays nearly constant over 2 core up to 64 where it takes approximately 45,000 CPU cycles, as opposed to the nearly 700,000 for 32 cores when regular page tables are used.

[2]At the time of writing this paper, the exact number of CPU cores on the Pre-Production Intel® Xeon Phi™ is confidential and the authors are under non-disclosure agreement.



(a) Hyper-threading disabled



(b) Hyper-threading enabled

Fig. 7: **Stencil computation runtimes.** *XGBv-YGBp denotes XGB application virtual memory and YGB physical memory. PSPT stands for partially separated page tables, pre-fetch indicates that dedicated data movement cores are also utilized.*

C. Stencil Computation

To assess the cost of OS supervised data movement we measured runtimes of the above described stencil computation kernel with various configurations. Figure 7 summarizes the results. We ran the application in all cases with 2 GB virtual memory allocated for the computation area. As the baseline for further comparison, we first measured runtimes with physical memory supplied enough so that no data movement takes place, that is to say, with physical memory also set to 2 GB.

As for configurations where data movement is also required, we limited the amount of physical memory to 1GB and we have three different setups. First, synchronous data movement (i.e., data movement is performed exclusively in page fault handlers) with regular page tables. Second, synchronous data movement with PSPT, and third, PSPT combined with dedicated data movement cores that execute pre-fetch requests.

We have then two sets of measurements, both containing the above mentioned four scenarios, one with each application thread executing on separate CPU cores, and the other, where hyper-threading is utilized, shown in Figure 7a and Figure 7b, respectively. As it has been said earlier, the Pre-Production Intel® Xeon Phi™ co-processor we used features 57 CPU cores, each with 4 hyperthreads. Because we dedicate one hyperthread for each application thread for doing pre-fetch operations, we measured the separate CPU cores case up to 56 cores, and the hyper-threading case up to 128. Notice, that for 128 application hyper-threads there aren't enough hyper-threads left so that each application thread could have its own dedicated pre-fetcher, and for this case only we distribute 64 pre-fetchers.

First thing to point out, regular page tables provide expected scalability in case there are no address space modifications, i.e., when the entire data fits into the physical memory. However, when data movement is done transparently by the OS and thus, frequent address space modifications are performed, regular page tables impose severe overhead on scalability just over 8 application threads. On the contrary, partially separated page tables show similar speedups to the no data movement case. Furthermore, PSPT combined with dedicated data movement cores yield runtimes even closer to the no data movement scenario.

TABLE I: Cost of transparent data movement compared to the no data movement case (hyperthreading *disabled*).

Number of cores	2	4	8	16	32	56
PSPT	32%	47%	38%	44%	56%	93%
PSPT+pre-fetch	11%	21%	14%	16%	26%	46%

The exact cost of data movement as the ratio of additional runtime compared to the no data movement case is indicated in Table I and Table II, without and with hyper-threading, respectively.

TABLE II: Cost of transparent data movement compared to the no data movement case (hyper-threading *enabled*).

Number of cores	4	8	16	32	64	128
PSPT	36%	32%	32%	37%	45%	33%
PSPT+pre-fetch	5%	8%	6%	7%	6%	22%

As seen, the price of synchronous data movement without hyper-threading is 51% in average, while pre-fetching decreases the cost to an average of 22%. The highest overhead we observe is when running the application over 56 cores, where we identified the bottleneck as the contention on DMA engines, suggesting that DMA is fully stressed in this case.

Moving on to the hyper-threading enabled experiments, one can see that the cost of data movement is relatively smaller. An average of 35% without and 9% with pre-fetching, respectively. This is mainly because the hyper-threading execution doesn't scale as fast as separate CPUs (as seen runtimes for 4 threads are actually worse than for 2), leaving more space for the background data movement. With all application threads having their own dedicated pre-fetcher, the price of data movement can be kept as low as 6% in average up to 64 application threads.

D. The Effect of Pre-fetching on the Number of Page Faults

The ultimate goal of OS assisted data movement is not only performing data copy in an on-demand, application transparent manner, but also to overlap computation and communication as much as possible.

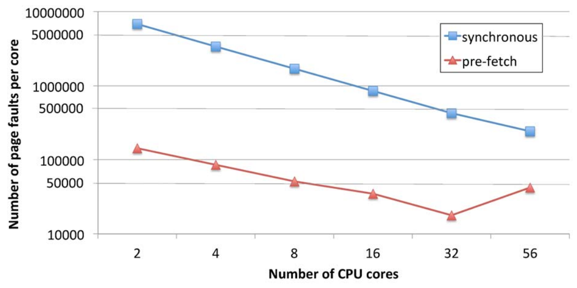


Fig. 8: **The effect of dedicated data movement cores on the number of page faults per CPU core.** *Hyper-threading disabled.*

Dedicated data movement cores serve exactly this purpose. As mentioned above, in our current prototype implementation, dedicated data movement cores execute pre-fetch requests on behalf of application cores. The effect of such activity can be well demonstrated by the number of page faults application cores cause. The less page faults there are, the more efficient the overlapping mechanism is. Figure 8 shows the per-core page fault numbers with and without dedicated data movement cores. (Note the logarithmic scale of the Y axis.) As seen, with background pre-fetching, the number of page faults is approximately two orders of magnitude smaller. It also shows good scalability with the increasing number of CPU cores, except for 56, where the tendency stops. As said in Section IV-C, pre-fetching for 56 cores causes contention on the DMA engine, which explains why the number of page faults fails to drop as fast as up to 32 cores, since pre-fetchers are unable to pull data ahead of the computation cores as fast as with smaller core numbers. Nevertheless, it still is an order of magnitude better than the synchronous case.

V. Related Work

A. Programming Models

Programming models for accelerators (i.e., co-processors) have been the focus of research in recent years. In case of GPUs, one can spread an algorithm across both CPU and GPU using CUDA [11], OpenCL [12], or the OpenMP [13] accelerator directives. However, controlling data movement between the host and the accelerator is the entirely the programmer's responsibility in these models.

OpenACC [14] 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.

Although in an accelerated system the peak performance includes the performance of not just the CPUs but also all available accelerators, the majority of programming models for heterogeneous computing focus on only one of these. Attempts for overcoming this limitation, by creating a runtime system that can intelligently divide computation (for instance in an accelerated OpenMP) across all available resources automatically are emerging [15].

Intel provides several execution models for Intel® Xeon Phi™ product family [16]. One of them, the so called *Mine-Your-Ours* (MYO), also referred to as *Virtual Shared Memory*, provides similar features to our proposal, 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.

As for memory models, 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. The asymmetry allows light-weight implementations that avoid common pitfalls of symmetrical distributed shared memory systems. ADSM allows programmers to assign data objects to performance critical methods. 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 [17]. While ADSM uses GPU based systems providing transparent access to objects allocated in the co-processor's memory, we are aiming at a symmetrical approach over Intel's MIC architecture.

B. Operating Systems for Manycores

Operating system organization for manycore systems has been also actively researched during the last couple of years. In particular, issues related to scalability over multiple cores have been widely considered.

Corey [9], 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 and to take advantage of the likely abundance of cores by dedicating cores to specific operating system functions. Similarly to Corey, we also focus on scalability issues of kernel data structures, namely, the page tables, and demonstrate that replication with careful synchronization scales better than centralized management. Nevertheless, our goal is complete application transparency. Moreover, similarly to Corey, we also dedicate certain OS functionality to specific CPU cores, i.e., background data movement, and show the benefits of doing so.

Barrelfish [18] 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 modern operating systems have been also the focus of recent research. Clements et. al [8] proposed increasing the concurrency of kernel operations on a shared address space by exploiting read-copy-update (RCU) so that soft page faults can both run in parallel with operations that mutate the same address space and avoid contending with other page faults on shared cache lines. Our approach also enables scalable updates to a shared address space, but instead of exploiting efficient locking heuristics on shared data structures we replicate data structures (page tables for the computation area) and allow page faults to update only the affected cores' replica.

An idea, similar to PSPT, has been discussed by Almaless and Wajsburt [19]. 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, no actual implementation is provided.

Villavieja et. al also pointed out the increasing cost of remote TLB invalidations with the number of CPU cores in chip-multiprocessors (CMP) systems [20]. 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.

VI. Conclusion and Future Work

In this paper we have presented an operating system assisted hierarchical memory management system targeting manycore co-processor based heterogeneous architectures. In our system, data movement between the host memory and the co-processor's RAM is controlled entirely by the OS kernel, so that data transfer remains completely transparent from the application's point of view.

We have found that with traditional address space wise page tables, OS assisted memory management on a manycore co-processor imposes severe performance degradation, especially, when address remappings are issued simultaneously on a large number of CPU cores. In order to overcome the scalability problem, we have proposed *partially separated page tables*, where each application thread (i.e., CPU core) maintains its own set of translations for the computation area of the application, filling in a virtual to physical mapping only if the given core operates on the particular address. This enables the OS to perform address space updates only on affected cores when remapping a virtual address to a different physical page, and thus, achieve good scalability. Furthermore, we have proposed leveraging idle CPU cores for dedicated data movement on behalf of the application. We have tested our prototype system on stencil computation, a common HPC kernel, and showed that OS assisted memory management is capable of scalable transparent data movement.

In the future, we intend to further address scalability issues for the case where a large number of cores are used. Also, because 2D stencil computation is a relatively straightforward application in terms of memory access patterns, we intend to apply our system to a wide range of other applications. We are planning to investigate how far the kernel can draw intelligent decisions on what data to pre-fetch and possibly offer APIs for the application to provide such information.

Acknowledgment

This work has been partially supported by the CREST project of the Japan Science and Technology Agency (JST).

We would like to express our gratitude to Intel Japan for providing the hardware, software and technical support associated with the Intel® Xeon Phi™ product family.

References

[1] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson, "Programming model for a heterogeneous x86 platform," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 431–440. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542525

[2] E. Saule and U. V. Catalyurek, "An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture," in *26th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP)*, 2012. [Online]. Available: http://bmi.osu.edu/ esaule/public-website/paper/mtaap12-SSC.pdf

[3] A. S. Tanenbaum, *Operating systems: design and implementation*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1987.

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

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

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

[7] A. Hori, A. Shimada, and Y. Ishikawa, "Partitioned Virtual Address Space," in *ISC'12: International Supercomputing Conference*, 2012.

[8] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Scalable address spaces using RCU balanced trees," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012, pp. 199–210.

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

[10] Q. Yuan, J. Zhao, M. Chen, and N. Sun, “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.

[11] N. Staff., “NVIDIA CUDA Programming Guide 2.2,” 2009.

[12] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 8 December 2008. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[13] OpenMP Architecture Review Board, “OpenMP Application Program Interface,” Specification, 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[14] CAPS Enterprise and CRAY Inc and The Portland Group Inc and NVIDIA, “The OpenACC Application Programming Interface,” Specification, 2011. [Online]. Available: http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf

[15] T. R. W. Scogland, B. Rountree, W.-c. Feng, and B. R. de Supinski, “Heterogeneous Task Scheduling for Accelerated OpenMP,” in *26th IEEE International Parallel and Distributed Processing Symposium*, Shanghai, China, May 2012.

[16] Intel Corporation, “Knights Corner: Open Source Software Stack,” 2012. [Online]. Available: http://software.intel.com/en-us/forums/showthread.php?t=105443

[17] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, “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*, ser. ASPLOS ’10. New York, NY, USA: ACM, 2010, pp. 347–358. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736059

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

[19] G. Almaless and F. Wajsburt, “Does shared-memory, highly multi-threaded, single-application scale on many-cores?” in *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism*, ser. HotPar ’12, 2012.

[20] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, “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*, ser. PACT ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 340–349. [Online]. Available: http://dx.doi.org/10.1109/PACT.2011.65