A Preliminary Study on the Design of Hierarchical Memory Management for Heterogeneous Architectures

Balazs Gerofi[1,a)] Akio Shimada[1,b)] Atsushi Hori[1,c)] Yutaka Ishikawa[2,1,d)]

Abstract: Heterogeneous architectures, where a multicore processor, which is optimized for fast single-thread performance, is accompanied with a large number of simpler, but more power-efficient cores optimized for parallel workloads, are receiving a lot attention recently. Currently, these co-processors, such as Intel's Many Integrated Core (MIC) software development platform, come with a limited on-board RAM, which requires partitioning computational problems manually into pieces that can fit into the device's memory, and at the same time, efficiently overlapping computation and communication. In this paper we explore the design considerations for operating system (OS) assisted hierarchical memory management, relying on the capabilities of the Intel MIC's memory management unit (MMU). We are aiming at transparent data movement between the device and the host memory, as well as tight integration with other OS services, such as file and network I/O.

Keywords: Operating Systems, Memory Management, Manycore Co-processor

1. 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 Many Integrated Core (Intel MIC) architecture is Intel's latest design targeted for processing highly parallel workloads. The current prototype Intel MIC cards, codenamed Knights Ferry (KNF), provides in a single chip up to 32 x86 cores, with each processor 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 MIC architecture 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.

The on-board memory is faster than the one in the host, but it is significantly smaller. A few Gigabytes on the card, as opposed to the 24 GBs residing in the host machine in our setup. 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.

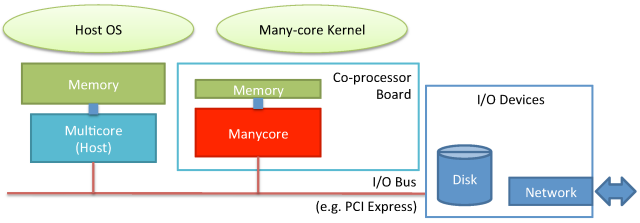

Fig. 1 Architectural overview of a multi-core host computer equipped with a many-core co-processor, connected through PCI Express.

However, the Intel MIC architecture features a standard memory management unit (MMU), which can be utilized to provide the illusion of having much larger amount of memory than that is physically available. Just like on the multicore host, the operating system is supposed to keep track of the physical memory and to manage the mapping from virtual to physical addresses. Thus, the OS running on the many-core unit can transparently move data between the card and the host, similarly how swapping is performed to disk in traditional operating systems.

Nevertheless, the case of many-core co-processor is different than regular disk based swapping. First of all, the host memory is substantially faster than a disk. Second, operations (such as disk or network I/O) on data residing in host memory can be directly performed on the host CPU without the need of moving the data

[1] RIKEN Advanced Institute for Computational Science
[2] Graduate School of Information Science and Technology The University of Tokyo
[a)] bgerofi@riken.jp
[b)] a-shimada@riken.jp
[c)] ahori@riken.jp
[d)] ishikawa@is.s.u-tokyo.ac.jp

back to the memory attached to the co-processor. Essentially, the co-processor's memory behaves as another level in the memory hierarchy. Third, there is a large number of cores available on the co-processor board, from which some may be utilized for monitoring execution, helping to make more intelligent decisions regarding data movement. How to address these issues at the OS level is the main focus of this study.

The rest of this paper is organized as follows, Section 3 outlines our envisioned execution model, Section 4 describes the memory layout of an application utilizing the hierarchical memory system, Section 5 discusses integration with I/O, and Section 6 describes our page replacement policy. Section 7 surveys related work and finally, Section 8 concludes the paper.

2. 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 many-core coprocessors.

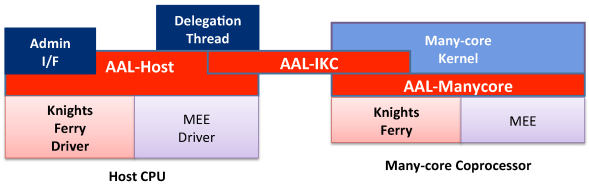

Fig. 2 Main components of Accelerator Abstraction Layer (AAL) and the many-core kernel.

Figure 2 shows the main components of the proposed software stack. The Accelerator Abstraction Layer (AAL) is designed to hide hardware-specific functions and provide kernel programming interfaces to operating system developers. The AAL resides in both the host and many-core units. AAL 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 many-core CPUs.

We have already explored various aspects of a coprocessor based system, such as scalable communication facility with direct data transfer between the coprocessors [3], possible file I/O mechanisms [4], and a new process model aiming at efficient intra-node communication [5].

We are currently developing a microkernel based OS targeting many-core CPUs over the AAL, 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.

3. Execution Model

In spite of the current architecture of a manycore co-processor based system, where the co-processor always comes with the presence of the 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. Thus, we are designing an execution model taking such transition into account.

Figure 3 depicts the execution model under consideration. 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. At present, we plan to provide such capability by means of C *#pragma*s, specifying the preferred target.

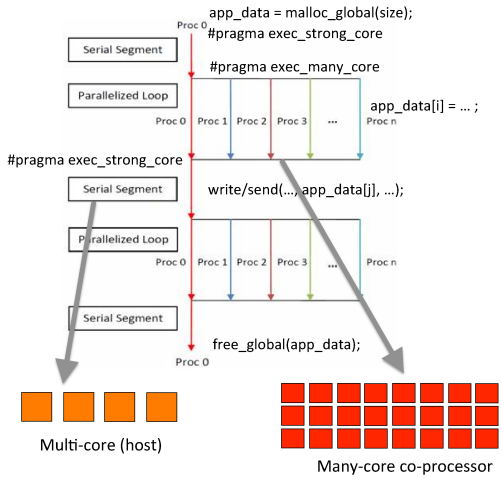

Fig. 3 Execution model with shared global memory.

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.

Furthermore, certain memory areas (which we call *global*) of the application are 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 provides these functions and ensures consistency when execution migrates between the co-processor and the host CPU. Mapping *global* memory areas to the same virtual addresses on the host and on the MIC would likely make it easier to implement the control of execution flow.

The usage of these functions is also demonstrated in Figure 3, where a global area is allocated in the beginning of the program. It is then consequently accessed in both the parallel and the serial sections of the code, while eventually it is freed by the corresponding library call.

4. 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.

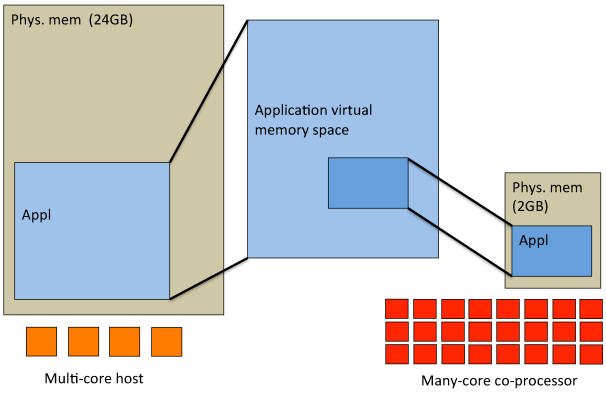

Fig. 4 Memory layout of an application running on the manycore co-processor and the multicore host.

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 can select victim pages and move the content to the host's memory. Also, when the execution flow migrates to the host, contents of the pages that still reside on the co-processor can be moved to the host. 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.

There are multiple ways how such system may be realized. One solution is that the host system's physical memory is accessed as a swap partition (e.g., through a network block device) in the kernel running on the co-processor using a standard communication protocol (such as UPD over IP). The advantage of this approach is ease of implementation. However, we believe there is a substantial software overhead (coming from the block device driver and the network protocol), which can be entirely eliminated.

Another way would be to memory map the host system physical memory as a file in the kernel on the co-processor using a lower level communication facility for accessing the host memory. We have opted to go with an approach similar to the one mentioned later, however, we are planning to bypass even the file abstraction layer.

In order to retain full control over the data transfer between the co-processor and the host, we will integrate 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 lower level modifications to the operating system organization, this way we are hoping we can eliminate any unnecessary software overhead both in terms of CPU consumption and additional memory footprint.

5. Integration with I/O

As mentioned earlier, one of the major differences between disk based swapping and storing data on the host memory is the availability of *swapped out* content on the host.

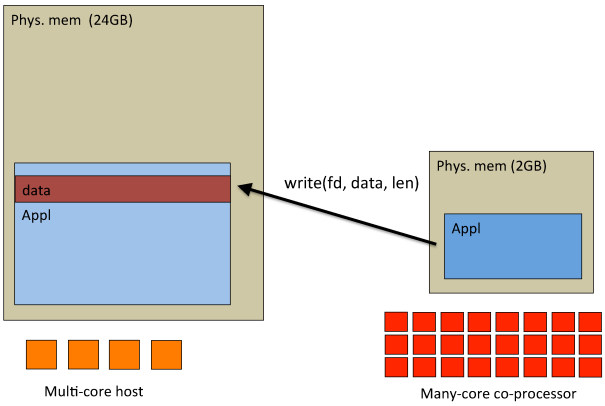

Fig. 5 Tight cooperation between the host and the co-processor for I/O operations.

This can be exploited in certain situations. For instance, imagine that the application running on the co-processor, which has part of its virtual address space stored in the host memory, issues a file I/O operation that refers to data residing in the host.

In a traditional swapping scenario this would cause the operating system to read the referred data into the main memory so that the operation can be carried out. However, in the case of multicore host equipped with manycore co-processor, tight co-operation between the kernels are possible. Because the kernel running on the co-processor has complete knowledge on which part of the virtual address space is currently available in the co-processor memory and which is located in host, it can request the host to processor the operation on behalf of the co-processor itself.

On the other hand, when the execution flow is located on the host and an I/O operation is invoked, the host needs to be able to update any data from the co-processor's memory which have been changed since the execution was last time on the host side. Clearly, this assumes that in certain system calls, such as *read()*, *write()*, or *send()*, extra care needs to be taken, regardless whether it is executed on the host or the co-processor.

For this purpose, a facility for tracking which page resides where is necessary both on the host and on the co-processor. Furthermore, this information will have to be carefully synchronized whenever data, or the execution flow is transferred.

6. Page Replacement Policy

In any demand paging system, one of the key challenges is the mechanism of page replacement. It is a widely accepted idea, that pages frequently used over a short period in the recent past are likely to be used in the (near) future. The *least recently used* (LRU) approach is based on the converse assumption that pages not used recently will not be needed frequently in the immediate future. LRU refers to various algorithms that attempt to find least used pages according to a similar scheme [6], [7].

While the fundamental LRU principle may be simple, it is difficult to implement it appropriately. The reason being traditionally, is that organizing data structures so that the kernel can mark or sort pages as simply as possible in order to estimate access frequency is a rather CPU consuming procedure. Therefore, current operating system kernels often opt for a coarse grained approximation of the LRU, attempting to alleviate the overhead.

However, in case of the manycore co-processor, where there are plenty of cores available, dedicating a core for performing statistical monitoring may be a feasible approach. In fact, recent OS research targeting manycore architectures already shifted from the temporal distribution of CPU cores towards a spatial approach, where certain cores are designated for taking care of specific system services [8], [9].

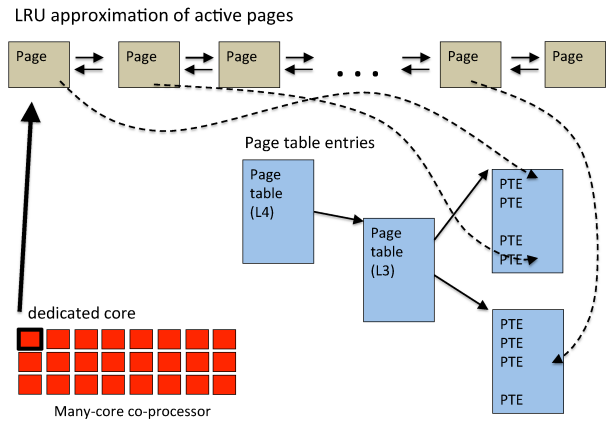

Fig. 6 Dedicated CPU core for fine-grained LRU approximation of page usage statistics.

Figure 6 illustrates the idea of tracking page activities by a dedicated CPU core, providing finer grained approximation of the LRU algorithm. A double-linked list of active pages is maintained, where each page structure holds pointers to the corresponding page table entries (PTE). The dedicated core periodically scans the LRU list, examines and clears the referenced bit of the PTE entry and updates the list accordingly.

When a page fault occurs and the physical memory is fully occupied, the page fault handler consults the LRU list in order to determine pages whose content can be moved to the host memory and can be reused for the faulting address.

It is also worth pointing out that it is desired to overlap data movement with computation on the manycore unit. The dedicated core for gathering page statistics can also initiate data movement (i.e., swapping out to the host memory) attempting to ensure that unused pages are always available whenever a page fault occurs.

A further optimization to this problem could be the involvement of the programmer herself. We are currently considering what an appropriate API could be for enabling the programmer to inform the OS in advance when certain memory area will be likely not used in the near future. This area could be then immediately transferred to the host, providing higher a chance for overlapping data movement with computation.

At the same time, the ability for prefetching data from the host to the co-processor is also essential. Again, similarly to the hints for swapping out content, we are planning to provide APIs so that such information can be passed to the runtime system.

7. Related Work

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 [10], OpenCL [11], or the OpenMP [12] accelerator directives. However, controlling data movement between the host and the accelerator is the entirely the programmer's responsibility in these models.

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 [13].

Intel provides several execution models for its MIC software development platform [14]. 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.

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 [15]. 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.

Operating system organization for many-core 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 [16], an OS designed for multi-core 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 intend to dedicate certain functionality to a specific core so that page usage can be tracked with better accuracy.

Barrelfish [17] 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.

GenerOS [9] partitions CPU cores into application core, kernel core and interrupt core, each of which is dedicated to a specified function. Kernel cores run several kernel servers, which are serial processes that provides a specific function for applications, such as a specific type of system calls. A kernel server always resides in kernel mode, therefore no kernel/user switch happens. Again, the idea of having utilizing dedicated cores for system call execution is similar to the utilization of idle cores in a manycore system.

Memory management in traditional operating systems, such as various algorithms for page replacement in a demand paging system has been widely considered in the literature [6]. For example, Linux implements a coarse grained LRU method [7], using two page lists, *active* and *inactive*. To distribute the pages between the lists, the kernel performs a regular balancing operation that determines whether pages are active or inactive, by means of the accessed bit of the corresponding PTE. As opposed to Linux' coarse grained approach we plan to utilize a spare CPU core so that better approximation of the LRU algorithm can be achieved.

8. Conclusion and Future Work

In this paper we have proposed an execution model together with a hierarchical memory management system for upcoming co-processor equipped configurations, targeting in particular the Intel Many Integrated Core (MIC) development platform.

The proposed execution model allows programmers to provides hints to the runtime system in order to express their preferences where certain parts of an application should be executed (i.e., on the host CPUs or the manycore co-processor).

We described a hierarchical memory management system that transparently moves data between the host and the co-processor, enabling programmers to focus on the computational part of the problem instead of dealing with data movement issues. The foundation of our approach is the memory management unit (MMU) present on Intel MIC, which allows us to provide much larger virtual memory than the memory physically attached to the co-processor.

We have described our approach to realizing such system with several possible optimizations, such as integration with I/O services, or the utilization of spare CPU cores on the manycore unit for achieving better page replacement performance.

In the future, once we finish the implementation, we are planning to carry out rigorous performance evaluation of the proposed system over various scientific applications.

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 MIC architecture.

References

[1] 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, *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, New York, NY, USA, ACM, pp. 431–440 (2009).

[2] Saule, E. and Catalyurek, U. V.: An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture, *26th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP)* (2012).

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

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

[5] Hori, A., Shimada, A. and Ishikawa, Y.: Partitioned Virtual Address Space, *ISC'12: International Supercomputing Conference* (2012).

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

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

[8] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A. and Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems, *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, New York, NY, USA, ACM, pp. 29–44 (2009).

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

[10] Staff., N.: NVIDIA CUDA Programming Guide 2.2 (2009).

[11] Khronos OpenCL Working Group: *The OpenCL Specification, version 1.0.29* (2008).

[12] OpenMP Architecture Review Board: OpenMP Application Program Interface, Specification (2008).

[13] Scogland, T. R. W., Rountree, B., Feng, W.-c. and de Supinski, B. R.: Heterogeneous Task Scheduling for Accelerated OpenMP, *26th IEEE International Parallel and Distributed Processing Symposium*, Shanghai, China (2012).

[14] Intel Corporation: Knights Corner: Open Source Software Stack (2012).

[15] 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, *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, New York, NY, USA, ACM, pp. 347–358 (2010).

[16] 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, *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, Berkeley, CA, USA, USENIX Association, pp. 43–57 (2008).

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