Memory Management on Chip-MultiProcessors with on-chip Memories

Carlos Villavieja*,Isaac Gelado*,Alex Ramirez*†,Nacho Navarro*
*Departament d'Arquitectura de Computadors. Universitat Politècnica de Catalunya
C/ Jordi Girona 1-3, Campus Nord. 08034 Barcelona, Spain.
e-mail: {cvillavi,igelado,aramirez,nacho@ac.upc.edu}
†Barcelona SuperComputing Center, Spain.

***Abstract*— In this paper we study the OS services requirect to efficiently manage on-chip memories in CMPs. CMPs typically include serveral cores connected to on-chip local memories. This architecture presents new challenges to the OS. Local memory elements in CMPs can act as cache memories or as local storage memories. When acting as local storage these local memories can be directly accessed by any core in the system using regular load/store instructions, without requiring any coherence action. Applications might use local memories to contain frequently accessed data. This usage pattern reduces the number of accesses to the main memory and the coherence traffic, improving the system performance.**

The OS is the software that abstracts and multiplexes hardware resources. Thus, applications relay on the OS to schedule the code to the core attached to the local memory where its data is stored. In this paper we discuss that a single address space to ease the programming of applications in heterogenous CMP systems. We also describe the necessary OS services to efficiently manage a single address space heterogenous CMP.

I. Introduction

The emerge of Chip-MultiProcessors (CMPs) and many-core chips has introduced huge computational power. The number of hardware threads per processor is increasing fast, however, traditional programing models, compilers and operating systems are still not able to take full advantage of it. Applications require of higher degrees of parallelism to use all the hardware threads CMP provide. Therefore, CMP processors require of more adequate programing models.

New programming models are already targeting CMP. Cell superscalar [2], Transactional memory, OpenMP and many others models are being used to make CMP programming easier. However, achieving a higher degrees of parallelism is not the only challenge, but to efficiently manage all the hardware resources CMP provide. CMP are also introducing on-chip memories. On-chip memories can improve applications performance by reducing memory access time. Some CMP include on-chip memories in terms of local memories or Scratch-Pad memories.

One example is the Cell BE processor. The Cell BE processor integrates many processing elements and many different memories on-chip. Besides traditional off-chip global memories and cache memories, the Cell integrates local on-chip memories also named Local Stores. Local Stores are included as non-cacheable distributed memories with guarantied latency and high bandwidth. A key difference between Local Stores and cache memories is that they are addressable. The programmer can explicitly manage data transfers from off-chip main memory to Local Stores and vice-versa. Each Local Store is only addressable with ld/st instructions from its local core. However, some CMP integrate on-chip memories directly addressable from any core.

For some parts of an application, cache memories may perform better than Local Memories(LM) and vice-versa. This is why Scratch-pad memories and Hybrid cache memories are also being introduced [1], [5] as Local Memories.

CMPs are not only being used in high performance computing but also in embedded computing as home gateways, game consoles or video players. These platforms are usually deployed with many streaming applications. Stream computing [6] is becoming one of the major target for CMP architectures. The main reason is the easy split of these applications in computation and communication kernels. Computational kernels are parts of an application working on a limited data set. We consider computing kernels as *block algorithms* because programmer can detected them in her code by blocks. These blocks compute massively a limited data set. Usually code loops operating on data vectors. *Block algorithms* are ideal to exploit SIMD capabilities of some CMP cores and the guarantied latency of on-chip memories. Mapping these data vectors to on-chip memories introduces important performance gains. Data access time is considerably reduced due to the guarantied latency and high bandwidth of on-chip memories.

As shown in Figure 1, on-chip memories are considered local memories. In this paper, we consider Local memories are addressable by any core in the CMP. For this reason, an application may use all on-chip and off-chip memory as a single address space. All data and code is accessible within a process virtual address space. With a load/store instructions, any core can access any data. On-chip data transactions have a much higher bandwidth than off-chip operations. Minimizing off-chip data transactions versus on-chip improves significantly applications performance because of the higher bandwidth between on-chip elements. This is why a new major goal is to provide efficiency in data allocation and placement.

This work has been supported by the Ministry of Science and Technology of Spain and by the European Union (FEDER) under contract TIN2007-60625, the HiPEAC European Network of Excellence (IST-004408) and the SARC European Project(EU contract 27648-FP6). All products or company names mentioned herein are the trademarks of registered trademarks of their respective owners.

We introduce using a single global name space in CMPs. Global name space allows any core to access any memory location using regular load/store instructions. A single address space accessible from any core makes easier to minimize off-chip traffic. As an example, data allocation in CMP also affects operating systems design. Inter-Processor communication [4] can be a lower cost operation using on-chip memories than architectures without on-chip memories.

In this paper, we make a design space exploration of the operating system memory management in CMP architectures. We study the mechanisms required to minimize off-chip main memory accesses. We study how data allocation and placement can deal with memories with different access latencies. We study how shared memory models are easy to program. We also study how to manage naming in CMP architectures, where the number of on-chip memory and processing elements may vary from one CMP version to another.

The main contributions of this paper are:

- A study of the usage of a single address space/shared memory model in CMP architectures.
- A study of static and dynamic memory allocation and placement in CMP architectures.
- A description of the implications of on-chip local memories in applications lifetime.

The rest of this paper is organized as follow: In Section II we describe related work in the area, Section III describes a general CMP architecture, Section IV describes the motivation for a single global name space for CMP, Section V presents the OS memory management services affected, Section VI presents some general implications of the memory management in the OS, and in Section VII we conclude.

II. Related Work

Most previous research in shared memory programing models has focused for Symmetric Multiprocessing architectures. Monchiero et al [10] makes a design space exploration in the energy/delay tradeoff on a shared memory architecture based on a NoC. The paper introduces using on-chip memories for shared programming models. However, memory management of many on-chip memories is not treated.

Mckenney [9] introduces how to perform performance estimations on shared-memory multiprocessors. His study can be also extended to CMP.

Cho et al. [3] investigates an OS-managed data to cache slice mapping. Based on a data proximity level and without hardware support they are able to efficiently manage the usage of L2 cache. This work is relevant in how OS memory allocation can affect application performance due to cache usage, however, we study this effect for on-chip addressable memories.

III. Inside the Chip-MultiProcessor architecture

In this section, we describe a generic scalable CMP architecture. The processor in Figure 1 will be our example processor architecture to illustrate all the OS services required. CMP architectures are usually composed by a set of processors

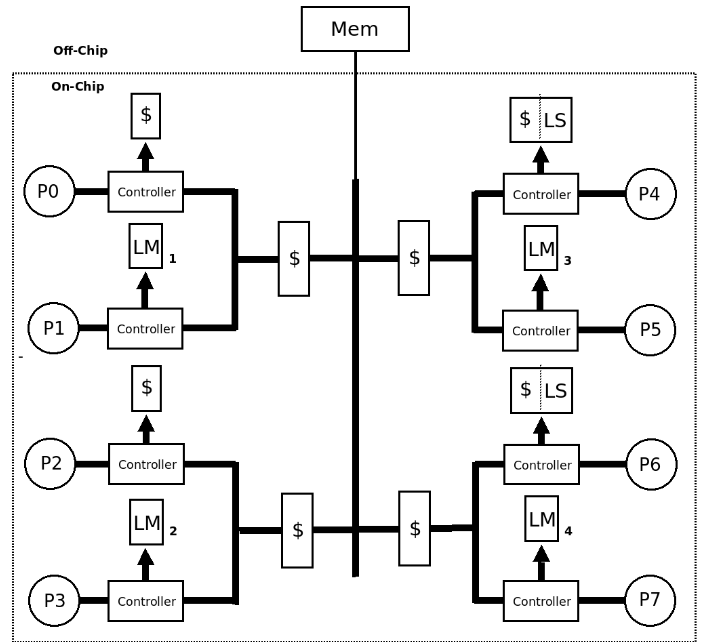


Fig. 1: Generic CMP architecture. The processor architecture is composed by eight processors or cores (Pi), and a set of local memories. The figure also n-chip shows the connection to main memory.

and local memories interconnected. Besides these on-chip elements, the off-chip main memory is also shown.

The following subsections describe the different memory elements of the architecture found in Figure 1.

A. Memory

Memory elements in a CMP architecture can be found on-chip and off-chip. The off-chip main memory is usually a DDR chip interconnected through a bus to the processor chip. The on-chip memories are connected to the cores through the on-chip interconnection network. Every memory operation is managed by a memory controller attached to the core executing the instruction. The controller is the responsible of locating the data in the physical memory.

Depending on the CMP configuration, on-chip memories may have a defined behavior. Some processor configurations may use on-chip memories as a second level cache, as a local addressable memory or even both. The following list is a description of possible behaviors of local on-chip memories:

- Cache memory ($): faster small memory to reduce the average time to access memory. Managed by hardware, typically a L2 cache memory. Cache memories are not addressable and they do not affect the OS memory management. Cache memories can be connected in a CMP for private use by one core or to shared use by various cores.
- Hybrid cache memory ($ —LS): Simple array of memories directly addressable and explicitly managed by software. When accessing it, there is no possible miss and its access time is predictable. Hybrid cache memories are

reconfigurable and they can act as an addressable memory or as a hardware cache memory.
- Local Memory (LM): Scratchpad memory, represents a local memory managed by software with a guarantied latency. These memories are smaller than hardware cache memories, consume less energy per access.They are directly addressable.

The operating system can only manage addressable memories by static and dynamic allocation. The operating system is not responsible of managing cache memories. The OS can only flush then cache content when a context switch occurs. Depending on the page allocation, it can also apply some policies for L2 cache management [3].

In order to efficiently manage memory allocation and placement, each memory operation has to be considered. Our memory management study for CMP focuses on taking into account memory latency time when allocating data. The memory latency of each physical memory (on-chip/off-chip) is required to evaluate memory allocation efficiency. For this purpose, at table I, a summary of all memory operation costs is shown approximated. The costs shown in the table, are calculated based in a CMP architecture with cores running at 3.2GHz. We consider an off-chip main memory latency of 300 cycles. Any memory operation from a core to its local memory is considered to cost a maximum of 20 cycles. Memory operations from a core to other cores local memories are considered to cost a maximum of 40 cycles. The interconnection network bandwidth between local memories (33.6GB/sec) and main memory bandwidth (10GB/sec), we consider approximated data from the Cell BE processor [7],[8].

Approximated Operation costs			
Operation	**Location**	**Description**	**Cycles**
Main Memory (MMa)	off-chip	DDR memory	300
TLB hit	on-chip	hit in the local TLB	1
Page Table (PTa)	off-chip	2*MMa	600
TLB miss	off-chip	1(TLB) + Page Table	600
Local Memory (LMa)	on-chip	TLB + LM	20
Remote LM (RLMa)	on-chip	TLB + LM + Network	max 40
TLB inv. broadcast	on-chip	Network	max 50

TABLE I: Cost of memory operations on/off-chip in a CMP architecture.

For the architecture in Figure 1, we will consider 50 cycles for any operation concerning an inter-core message protocol.

IV. Global Name Space

Research in programming models for CMP architectures has rapidly increased in the last years. Newer programming models try to achieve higher degrees of parallelism in applications, and make CMP programming easier. However, programming models require of flexible abstractions from the lower software levels. Dealing with partitioning of applications data and code makes programming hard.

In CMP architectures, on-chip memories present the challenge to integrate them as private or shared memories. Private on-chip memories require the programmer to transfer data and code from off-chip main memory to on-chip memories and vice-versa. On-chip memories are much smaller than off-chip, and therefore the programmer must deal with partitioning its application.

Shared memory models allow an application to access all its virtual address space using load/store instructions. Therefore, the programmer does not have to transfer data explicitly between memories. This makes CMP programming easier.

However, a programmer may require to map certain data into on-chip memory to improve its application performance. This discloses the problems we try to solve:

- How to identify each on-chip memory from any core in a CMP processor.
- How to allocate the more used data at the smallest available latency on-chip memory.

To solve these problems, we present a global name space. The following are the main characteristics a global name space can explore:

1) Shared memory model. Applications can access all physical memory space from any CMP core. Applications can allocate data at a certain on-chip memory with a well known latency. Therefore, application can improve their performance by allocating most used data in low latency memories.
2) Swap data between memories. Swapping data between on-chip and off-chip memory can also benefit application performance. Off-chip main memory can be used as a secondary storage.
3) Load/Store instructions to access all memory space. Simple memory instructions from any core can read/write from/to any available physical memory. No address space switch is required.
4) No data replications. No coherence protocol is required by hardware in local memories. Cache coherence protocol are a well known limitation for processor scalability. Application control where data is stored.

V. Memory Management

The processor Memory Management Unit (MMU) allows to map a process virtual address space onto the physical memory. The virtual address space provides a unique view of the address space for all processes. Having a virtual address space allows to run applications with greater memory demands than existing physical memory. At the same time, it offers protection and facilities for data sharing. A virtual address space also allows to run multiple applications simultaneously in the same processor. At link time, the linker maps data and code through all the virtual address space. At load time, each program loaded has a complete view of the virtual address space for itself. The loader allocates the program onto physical memory. Once a program is loaded, all the memory operations are done through load/store operations or through library and system calls (allocation).

In CMP architectures with directly addressable on-chip memories, the OS needs to identify each physical memory to efficiently map applications data and code.

In this section, first, we study the load/store instructions behavior in a CMP processor. We describe the mechanisms

required for load/store instructions for any physical memory. Second, we describe how memory is allocated, and how data is placed for any application. We also describe how an application can manage memory latencies when allocating.

A load/store instruction can access any physical memory within the CMP processor architecture described in Section III. The following are the steps done for a memory access:

1) The processor at Figure 2 requests a logical address. A logical/virtual address request is sent to the processor cache (L1 allocating virtual addresses) and the TLB. In case of hit in the L1 cache memory and the TLB, the processor gets the data from the L1 cache memory, and continues executing the next instruction. In case of a L1 miss, the address page tag of the logical address is compared at the TLB in order to make the translation from logical page to physical frame and move towards step 2. A hit in the TLB means the translation for the page is valid, and it also validates the L1 cache hit. In case of a miss in the TLB, an access to main memory (off-chip) is done to read the corresponding page table entry. In Linux OS, a page table access is two/three (if PAE activated) memory accesses. Once the page table is accessed, the translation page-frame is updated in the TLB. With the translation of the page, we get the physical address and we move towards step 2.
2) Once the logical address is translated to physical at the controller, the controller is responsible to find the data for that address in all physical memory. The physical memory address space contains all on-chip and off-chip memory. The physical address is compared in the Controller to locate its memory bank. The physical address is the method to identify each local memory. In case it is a local address, the physical address is requested to the local memory bank. A request from a core to its local memory has a low latency. (Table I). In case the physical address is remote, the data request is sent to the next memory hierarchy (step 3). Some processors do not have a local memory, but a second level cache (L2). L2 cache memories are not addressable. If a core in the CMP has a L2 cache memory, data request is replied from the L2 in case of hit (skip off-chip access). In case of miss, the request is sent to the next memory hierarchy level.
3) Remote requests can either be to a off-chip main memory or to a remote on-chip local memory. The search of the physical node containing the data is a request to the next level of the memory hierarchy through the interconnection network.

Besides how memory instructions are managed by the processor architecture, memory allocation is key to efficiently manage the processor resources. In the following subsections, the memory allocation mechanisms are studied. Memory allocation can be divided in two areas: at application execution (dynamic) and at load time (static).

A. Dynamic memory allocation

Dynamic memory allocation is the assignment of a part of memory during the runtime of a program. During an application execution, it may require extra memory to continue executing. As application request more memory, the operating system tries to find a suitable unused memory chunk. Application computation may happen in one or more cores. Load/Store instructions from a core may access data allocated locally, remotely at a local memory or at main memory. As an application may be computational distributed, the application working set should be also distributed along the different local memories to minimize its access time. Memory allocation requests may require explicitly memory latency. This way, the programmer is able to allocate most used data at a low latency and improve its application performance.

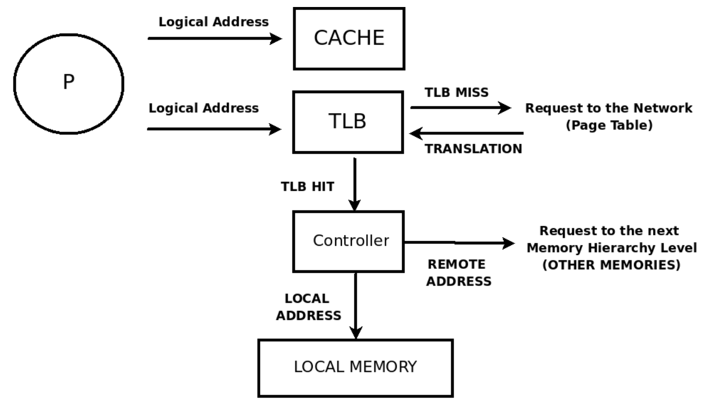


Fig. 2: Hardware scheme for any memory instruction. Load/Store instructions steps through the memory hierarchy. Any memory access is requested to a L1 cache, and if it is a miss, to the next memory hierarchy. The next level is a Local Memory. If the address of the request is not local, the request is sent to the next memory hierarchy through the on-chip network.

The main memory allocation functions are as follows:

- `void * malloc(thread_id tid, size_t size, [long latency,bool mandatory])`: This function allocates of a chunk of memory in the physical space of the requesting process. The owner of the chunk is the requesting thread, unless it is different from the *tid* parameter. The memory allocator tries to place the allocated chunk at a low memory latency from the core where *tid* thread executes. Using this function, an application can request the allocated memory chunk to be explicitly at a certain memory latency of the thread. The *latency* parameter allows the application to establish the maximum access time to access this data. The memory allocator tries to find a free space in any of the physical memories that accomplish the latency requirement. The policy for this, it is to place the requested chunk in the furthest memory that accomplish the latency access time request. We believe using this policy minimizes the waste of on-chip memory. On-chip memory is fast memory but smaller than off-chip. If the latency time parameter request is not possible to accomplish, the memory allocator can return an error. Returning an error or continuing execution depends on

the *mandatory* parameter. If *mandatory* parameter is set and the malloc function fails an error is returned. If the *mandatory* parameter is not set, the data is placed at the most appropriate memory for the latency time required. *Latency* access time and *mandatory* are optional parameters.

- `int re_map(void * src_logaddr,void * dst_logaddr,size_t size, [long latency,bool mandatory])`: Move a chunk of data to another address. Source and destination address may be at different physical memories. Remapping data function also provides *latency* and *mandatory* parameter to allow an application to transfer some data to a memory area with a different latency. An application may require to move a piece of data from one memory address to another in order to have data near its computation. An example of this can be a thread migration.

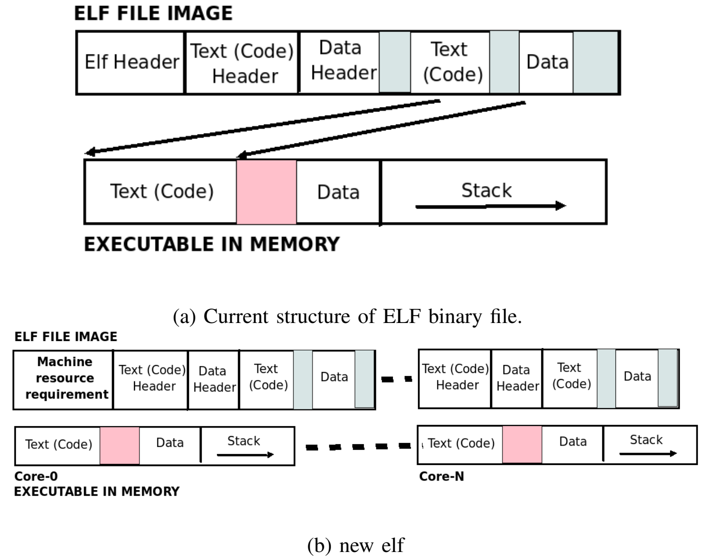
B. Static memory allocation

Static memory allocation is the assignment of a part of memory at load time. The loader is the responsible to load programs from binary files to memory. The loader gets information from the program to load from its ELF binary file. An ELF binary file has a well defined structure based on sections. Among all the sections, the most relevant are: Text, Data and Bss, as shown in Figure 3a. Text section contains the application code, Data section contains all initialized program's data, and Bss section contains all uninitialized data. At load time, the ELF binary file of a program is parsed and the loader allocates code and data into memory. When it allocates the code it gets a memory address from the ELF as the entry point of the program. The entry point is the start address of the program. However, in CMP architectures the loader may be able to allocate the different sections in main off-chip memory and on-chip local memories. The new ELF header shown in Figure 3b includes a description of the hardware requirements to run the application. These requirements are described in the *machine resource requirement* section file. The rest of the ELF file sections, specify code and data to be loaded on the local memories and main memory of the CMP. An application may provide more than one entry point. Therefore, at application start, more than one thread could be already created. This allows applications to efficiently use all core from application start. However, to maintain binary compatibility, the same binary should execute in different CMP architectures.

The *machine resource requirement* section should be checked at application load time. When loading the program, the loader must check if the CMP architecture fulfills the hardware requirements. This section might provide a minimal hardware description of the processor required resources (hardware threads, memory latencies, etc).

VI. General Implications

We showed in Section V that having on-chip memories has a performance impact in memory allocation and placement. In this section, we describe the impact of memory



(a) Current structure of ELF binary file.

(b) new elf

Fig. 3: Modifications to the ELF binary file format, code and data for each processing element of the CMP are introduced. The first section of the file is a set of requirements to accomplish in order to successfully run the application.

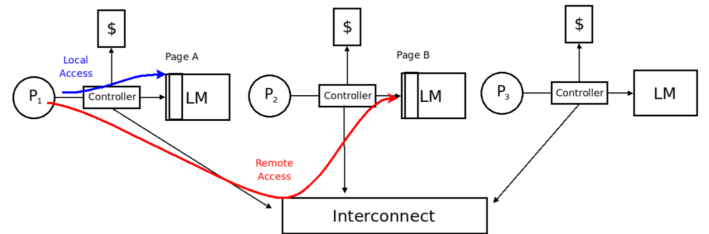


Fig. 4: Page migration from a core local memory to a remote local memory. Besides the allocation on both memories of the page chunk, a TLB invalidate protocol has to be done previous to the data page transfer.

latency in other operating system services. On-chip memories latency, as shown in Table I, is significantly less than off-chip main memory. Therefore, memory management in CMP must deal with memories latency when allocating data for applications. An application performance may vary depending on different factors from process priority to processor available resources. In order to make an efficient memory allocation policy, memory latency becomes a first class parameter for the operating system. Besides, memory allocation, several operations have to be taken into account. The following is a set OS services/operations to consider:

- Page Migration: A memory page stored in a local memory, as shown in Figure 4, may be binded to a thread and to an access memory latency. When the operating system migrates a page from one memory bank to another, it must check that the destination memory where allocating the page accomplishes the access latency required by the thread when the page was allocated. Once the destination memory is chosen, the TLBs of all cores containing the tag page must be flushed. This must be done because

the physical address of the page changes. TLBs must be flushed in order to invalidate the old page location. Therefore the page table must be updated for the new physical address where the page is moved. Special attention has to be paid to maintain coherence in the TLBs update process. The following actions should be done:

1) The source core invalidates a page table entry within its own TLB.
2) The core issues a TLB invalidated notice.
3) The core broadcasts the notice to other places.
4) The other cores search their own TLB for the notified entries.
5) The other cores invalidate entries of their own as notified.
6) The other cores issue acknowledgements.
7) The originating core issues a synchronization notice to other cores.

In order to emphasize the importance of memory placement, we calculate an approximated cost of a page migration. We consider a 4kB page size. Two different migrations are compared in table II, from a local memory to a remote local memory and from local memory to main memory. The cost of the page migration are approximated. We consider using a CMP processor with the Cell processor costs[8]. Jimenez et al.[7] evaluate an exhaustive list of experiments to measure on-chip and off-chip memory operations. We use their effective bandwidth measurements from Local Memory to Local Memory (33.6GB/sec) and from Local Memory to Main Memory (10GB/s).

Approximated Page Migration cycles costs		
Operation	**LM to LM**	**LM to MM**
TLB access	1	1
TLB inv.broadcast	50	50
Effective Bandwidth/sec	33,6GB	10GB
4Kb data transfer	390.1	1310.72
Total Cycles	440.1	1360.72
LM outperforms x3.1		

TABLE II: Approximated cost of a page migration from a local memory to a remote local memory and from local memory to main memory in a CMP architecture.

- Thread Migration: An application composed by one or more threads might require to migrate a thread from one core to another. A thread can be described by an execution context, a set of code and data pages. When a thread is migrated, the operating system may transfer all thread's code and data from one core to another. Since all local memories are addressable, the memory pages can remain in the source processor local memory. However, all migrated data require TLB flush since the physical address changes. Once the thread is stopped and a destination core is free, the thread context has to be moved to the destination core. Besides the thread context, and the source core TLB flush, the local memory pages should be moved to the new local memory in the destination core. For each page migrated, the page table also requires to be updated.
- Memory swap management: A CMP processor with on-chip and off-chip memories has a large memory hierarchy. The CMP memory hierarchy of these processors makes application performance dependent of data layout. Since local memories are smaller and faster, large data chunks must be stored in larger memories, usually off-chip memory. Therefore, the off-chip memory can be seen as a secondary storage and apply swap policies between on-chip and off-chip memories. Whenever local memories lack of space for new data, main memory can be used as a swap memory and only transfer data to local memory when it is used.

VII. Conclusions and Future Work

In this paper, we have studied the memory management for new Chip MultiProcessors (CMPs). Future CMPs integrating many cores and memories require of efficient mechanisms from the operating system to allow programming models, compilers and programmers to maximize application performance. CMP introduce on-chip addressable memories in order to improve memory performance. Memory access time varies considerably from on-chip to off-chip memories and the OS has to efficiently manage memory allocation and placement. Memory latency and bandwidth are key for the performance of many algorithms and applications. In this paper, we have shown how using a global name space, the operating system can manage the different memories. Using a global name space, different physical memories can use physical address to identify on-chip memories. We have also enumerated a list of advantages of a shared memory model for programming models. We have also shown that having access to the whole address space with load/store operations makes easier efficient programming.

The increasing relative latency to memory, taking hundreds of cycles (off-chip), means that applications must be aware of on-chip memories if they want to perform well. For this reason, we have introduced latency as a first class parameter in OS memory allocation functions. Using a global name space, programmers can manage application working set when allocating data and explicitly require latency access time. This and a shared memory model makes CMP programming easier.

In the future, we would like to investigate in more detail how these on-chip memories impact the operating system and how the operating system can take advantage of them. We are building a full system simulator and modifying Linux kernel to support the functionalities mentioned in the paper and make a exhaustive quantitative evaluation.

References

[1] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proc. of the 10th International Workshop on Hardware/Software Codesign, CODES, Estes Park (Colorado)*, May 2002.

[2] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *Proceedings of the ACM/IEEE SuperComputing 2006 Conference*, November 2006.

[3] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[4] I. Gelado, J. Cabezas, L. Vilanova, and N. Navarro. The cost of ipc. In *Second Workshop on the Interaction between Operating Systems and Computer Architecture WIOSCA'07, In conjunction with ISCA-34*, 2007.

[5] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J.Dally. Architectural support fot the stream execution model on general-purpose processors. In *Proceedings of the Sixteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[6] J. Gummaraju and M. Rosenblum. Stream programming on general-purpose processors. In *Proceedings of IEEE International Symposium on Microarchitecture (MICRO 05)*, 2005.

[7] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez. Performance analysis of cell broadband engine for high memory bandwidth applications. *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 210–219, 25-27 April 2007.

[8] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26(3):10–23, May-June 2006.

[9] M. P. Mckenney. Practical performance estimation on shared-memory.

[10] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Exploration of distributed shared memory architectures for noc-based multiprocessors. *J. Syst. Archit.*, 53(10):719–732, 2007.