Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

A control-inspired iterative algorithm for memory management in NUMA multicores

Marcello Farina [*] **Davide Zoni** [**] **William Fornaciari** [*]

[*] *DEIB, Politecnico di Milano, Milano, ITALY*
(e-mail: {marcello.farina, william.fornaciari}@polimi.it)
[**] *DEIB, Politecnico di Milano, Milano, ITALY*
(e-mail: zoni@elet.polimi.it)

Abstract: The multiprocessor revolution allows to have different processors accessing their local memory controller and the relative RAM portion, while the possibility for each processor to access the remote memory on another processor enables the so-called Non-Uniform-Memory-Access (NUMA) multiprocessor paradigm. This paper discusses a control-inspired iterative algorithm to allocate the memory requests by different applications in a NUMA systems, trying to maximize the RAM utilization under the locality requirement. The proposed method is tested in two case studies.

Keywords: Real-time algorithms, memory allocation, multicore systems.

1. INTRODUCTION

Memory management in modern operating systems (OS) exploits a paging mechanism, where the page is the small unit that can be allocated and deallocated in the main memory (RAM). When the number of required pages from all the running tasks exceeds the available RAM space, a swap-out process takes place to store the extra pages in the swap area, which is usually a portion of the hard disk. This mechanism allows the set of running processes to collectively access more memory than the available RAM. In particular, the memory management system is devoted to handling three different components: 1) the kernel allocator, which is the OS module that allocates and deallocates pages (eventually swapping them out); 2) the swap space, i.e., a portion of the hard disk in which to host the pages which exceed the available RAM; 3) the memory controller, i.e., the hardware component that manages the communication between the main memory and the processor.

The multicore revolution, fueled by the need for ever-increasing performance, forces the integration of multiple cores inside the same chip. It is worth noticing that the memory management systems are able to satisfy the memory allocation between multiple applications, in view of the presence of multitask supports. However, the use of multiple cores exacerbated the number of memory requests from tasks, thus single memory controllers become the bottleneck of the entire system. In this scenario, the multiprocessor solution emerges as a viable way to deal with the limited bandwidth imposed by the need to manage all the memory requests using a single memory controller. Multiprocessors are composed by multiple processors, where each processor is a multicore. They allow to split the physical memory between single processors, each endowed with a memory controller. For example, Figure 1 reports the Intel Nahalem multiprocessor, that is a 2 processor architecture, where each processor has four CPUs and a local Integrated memory Controller (IMC in Figure 1).

An additional issue in the multiprocessor design is imposed by the applications, which may require to share data, thus forcing

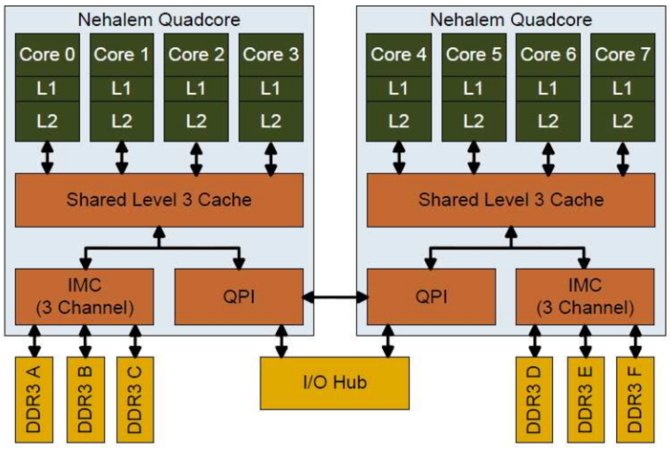


Fig. 1. Eight-core Intel Nahalem Processor (Molka et al., 2009).

the need of a shared memory space. This means that the main memory is physically split in banks, while it is a single memory space from the application viewpoint and each application can access all the banks, i.e. the local and the remote ones. The access to the remote memory banks is supported by cross-chip interconnects and the local memory controller of the remote multicore. Such a shared multiprocessor defines the concept of Non-Uniform-Access-Memory (NUMA) architectures, since the time to access a memory region depends on the location of both the requiring processor and the memory bank. In this scenario, the major processor design companies developed their own cross-chip interconnect to fill the NUMA gap, i.e., Intel QuickPath Interconnect (QPI) (Ziakas et al., 2010) and AMD's HyperTransport (AMD, 2002). Figure 1 shows the Intel cross-chip interconnect for multiprocessor interconnect (QPI in the figure).

In the NUMA multiprocessor scenario, data locality is a highly desirable property. Specifically, data should be kept close to the processor for which they have been allocated. In particular, the processor should always process data stored on the local memory node. However, due to shared memory areas and the possibility to require for more memory than that available on a single node, the allocation on remote node support is mandatory. Last, the memory access on a remote node is faster than the access to the swap area, since the latter requires a swap-in process from the hard disk, which is order of magnitude slower than the access to the RAM.

Memory contention due to multiple running applications represents a second issue to be addressed. While the total memory occupation is a system-wide metric, the user experience is based on the system responsiveness metric. For example, if a memory-intensive background task is running in parallel with a user-interactive application, the allocation of a huge memory space for the former can penalize the second one, due to excessive swapping action to disk, i.e. the main memory saturates and the memory manager swaps out according to an application-agnostic policy. This means that smart allocation and deallocation policies are required to manage memory, with particular emphasis on the portion which exceeds the available RAM space. Moreover, both data locality and the possibility that some applications can conflict with each other due to unbalanced memory allocation requests must be considered.

In this paper we adopt a control-inspired methodology to devise a dynamic iterative algorithm for memory management in the NUMA multicore scenario. We move a step further with respect to the state of the art in different directions:

I The proposed solution, based on a dynamic model of memory allocation, allows to reduce the bank-to-bank memory exchange and to maximize the local memory allocations to enhance the performance and minimize the power consumption. Furthermore, the proposed algorithm is extremely flexible, in the sense that it can be reconfigured by simply re-defining the neighboring relationships among memory banks.

II The inclusion of an *invasion/retreat*-type strategy in the control-inspired algorithm to optimally allocate memory and avoid inter-application memory conflicts. In particular, the proposal tries to maximize the allocation on local nodes first, while the memory can be allocated on remote nodes when the local bank is full. Moreover, the require for memory by the local processor can force the swap-out process, i.e. deallocation from main memory, of the memory store by remote processors.

The paper is organized in four sections. First, the state of the art related to NUMA multiprocessors is discussed in Section 2. The proposed algorithm is described in details in Section 3, while Section 4 discusses some simulation results. The future works and conclusions are presented in Section 5.

2. RELATED WORKS

The virtual memory technology enables the possibility to manage the process-allocated memory, where the sum of the memory required by all processes can be greater than the available physical RAM. This is possible thanks to the so-called swap process, which selects some pages from the main memory and deallocates them to the swap area through the so-called swap-out operation. In this framework, different solutions have been proposed to optimally swap-in and -out memory pages exploiting the lower access time for a RAM stored page with respect to swap stored one. The paper (Jones, 1969) is one of the first works proposing an on-demand page swapping procedure, i.e. when a page fault occurs the required page is swapped back into main memory (RAM). While different schemes mainly focused on swap-in and -out complete task memory have been proposed, the solution actually implemented nowadays is quite similar to the work in (Jones, 1969) proposed in late 1969. However, the work in (Terraneo and Leva, 2013) presents an interesting control-based methodology to solve the memory management considering an active swap-in policy which progressively swaps back pages from the swap area to the main memory depending on the available space in RAM without the need for a page fault. Moreover, a memory quote is associated to each application which represents the maximum available memory for the application in case of memory contention.

The works presented up to now are focused on Uniform Memory Access (UMA) designs and can not be applied to NUMA architectures (Blagodurov et al., 2010). In the latter, a discussion is given about the fact that contention management algorithms fail to be effective on NUMA systems and may even hurt performance relative to a default OS scheduler. In particular, the work outlines the need for remote memory access as a primary source for inefficiency. Starting from this observation, in the present paper we focus on dynamic memory allocation to fit variable memory requests minimizing the remote memory accesses.

The paper (Zuo et al., 2009) proposes a group-based hardware design for NUMA cache hierarchy to optimally allocate the data based on a locality policy. While the proposed solution exploits the data locality principle, it does not include a invade-and-retreat policy, which would allow for a flexible managing of the application memory requests which exceed the available local memory.

An algorithmic scheduling solution to maximize the data locality is proposed in (Majo and Gross, 2011). The focus of the work is on the cache and application scheduling to optimize cache accesses and data migration between different processors, while data migration represents the most challenging issue to be addressed to limit the remote access overhead. Conversely, our proposal focuses on the minimization for interconnect accesses between processor pairs to access a remote memory node, while ensuring a flexible memory allocation.

Finally, the work (Dashti et al., 2013) focuses on the memory controller congestion which hampers the performance. In particular, modern NUMA systems reduce the remote access time, while the contention on both memory controllers and interconnect still introduces non-negligible performance penalties. In this respect, the method proposed in (Dashti et al., 2013) does not consider the mutual influence on the memory allocation due to memory-bound tasks, which may force memory swap-out for different applications. Our proposal accounts for this aspect and limits the interconnect congestion promoting local memory placement.

3. PROPOSED METHODOLOGY

In this section we describe a novel iterative allocation (RA) algorithm proposed for the NUMA scenario. After introducing the main variables of the model of the memory allocation system, the underlying rationale of our scheme is presented. Then, the RA algorithm is described in details and application aspects are dealt with. In particular, since the RA algorithm is event-triggered, some comments are due on the hierarchical application scheme and, in view of its inherent iterative structure, we will briefly focus on the underlying communication/computation load required.

3.1 A model of memory allocation

We consider a NUMA multiprocessor, composed of M processors, and a set of tasks running on it. We also assume that each processor is endowed with a local memory controller and a memory bank. Therefore, for each processor we define a memory bank identifier $i \in \{1,\ldots,M\}$. For example, Figure 1 shows a two-banks NUMA multiprocessor, i.e., with $M = 2$. Moreover, we denote with T_i the set of tasks executed the i-th processor, and with B_i the i-th memory bank, $i = 1,\ldots,M$. We denote with variable x_{ij} the memory allocated by the taskset T_j on bank B_i. Correspondingly, $\mathcal{N}_i$ defines the neighborhood of memory bank B_i, i.e., the set of surrounding banks j (and therefore excluding i) whose tasks T_j can allocate memory in M_i. Therefore, x_{ij} is not identically equal to zero if and only if j is in the neighborhood set of i. We assume that x_{ij} is not identically equal to zero if and only if x_{ji} is not identically equal to zero, i.e., $j \in \mathcal{N}_i$ if and only if $i \in \mathcal{N}_j$.

We assume that the memory allocation event is initiated when memory allocation/deallocation requests are given. For generality, we assume that multiple requests for memory allocation/deallocation can reach the operating system (OS) simultaneously. Each request is related to a specific running task and requires a portion of main memory physically located on a bank. We denote with $r_i \geq 0$ the memory request from T_i (to be allocated, if possible, on bank B_i and/or on its neighboring banks B_j, $j \in \mathcal{N}_j$) and with $d_{ij} \geq 0$ the request of memory deallocation of T_j located on bank B_i where $d_{ij} \leq x_{ij}$ for all $i, j = 1,\ldots,M$.

We assume that, at a given time instant, each memory bank B_i, $i = 1,\ldots,M$ allocates for the taskset T_j, $j = 1,\ldots,M$, a given memory amount (denoted with x_{ij}^- for simplicity of notation) and that an allocation/deallocation request is given. Our scheme aims to select the amount of memory to be allocated by each task on each bank after the request. For simplicity of notation, we define with x_{ij}^+ the amount of memory allocated on memory bank B_i for the taskset T_j after the requests have been processed. We solve the re-allocation problem using the iterative RA algorithm described next.

3.2 The iterative allocation algorithm

After the "allocation/deallocation" event is triggered, the goal of our multi-core allocation procedure is many-fold:

I) Satisfy all the deallocation requests.
II) Satisfy locally the memory request, if possible. Indeed, if space is available, it is advisable to allocate r_i on bank B_i.
III) If the requests cannot be fully satisfied locally, try to allocate memory on the neighboring banks.
IV) If the requests cannot be fully satisfied locally and using the neighboring banks, allocate the required memory in the swap area.

Other requirements and constraints are also needed to complete the scenario.

i) The memory allocated on B_i cannot be greater than the capacity of B_i itself (denoted with x_i^{MAX}), i.e.,

$$\sum_{j\in\mathcal{N}_i\cup\{i\}} x_{ij}(t) \leq x_i^{MAX} \tag{1}$$

ii) Priority is given to local memory requests.
iii) To transfer memory load from one bank to another (or from the swap area to the banks) requires power. In view of this, if space is available on bank B_i at the end of step IV), we do not recall memory load from the swap area or from the neighboring banks. This can be done, in case, in a smooth and energy-saving fashion, after the results of the RA algorithm are applied to the system and before the occurrence of the next event, similarly to the approach proposed in Terraneo and Leva (2013).

First note that I) can be performed instantaneously (i.e., as soon as the event is triggered). Secondly, in view of the priority constraint (ii), in task II) we assume that all the memory allocated on the local bank B_i by other tasks T_j, i.e., x_{ij}, $j \neq i$, must be deallocated, if necessary (i.e., if task T_i needs memory space).
Task III) is then attained through a suitable negotiation-type iterative algorithm. Concerning this, and denoting with k the k-th iteration step of the latter, we define two fundamental quantities. First, the *available memory for external allocation on bank* i, i.e., the residual memory availability on each bank to host neighbor memory, at step k, is defined as

$$e_i(k) = x_i^{MAX} - x_{ii}(k) - \sum_{j\in\mathcal{N}_i} x_{ij}(k) \tag{2}$$

In (2), the term $x_i^{MAX} - x_{ii}(k)$ is the space in B_i, at iteration k, which is not used for satisfying local requests. The term $\sum_{j\in\mathcal{N}_i} x_{ij}(k)$, on the other hand, is the space allocated on B_i by the neighboring tasks to satisfy neighboring requests, at iteration step k.
Secondly, we define the *required memory not yet allocated from task* T_i at iteration step k as

$$n_i(k) = r_i - \sum_{j\in\mathcal{N}_i\cup\{i\}} d_{ji} - \Big(\sum_{j\in\mathcal{N}_i\cup\{i\}} (x_{ji}(k) - x_{ji}^-)\Big) \tag{3}$$

In (3), the term $r_i - \sum_{j\in\mathcal{N}_i\cup\{i\}} d_{ji}$ is the net memory request from the taskset T_i, while the terms $(x_{ji}(k) - x_{ji}^-)$, for all $j \in \mathcal{N}_i \cup \{i\}$ is the net memory allocated (note that it can be negative if deallocation requests prevail) on bank j by the taskset T_i since the beginning of the event.
Finally, the requirement IV) is attained after the latter iterative distributed algorithm converges to a steady-state.
Now we are in the position to present the RA algorithm.

Algorithm 1. **RA**

1. **Initialization**: the algorithm initialization (i.e., step $k = 0$) is performed in view of 1) and 2), i.e., for all $i = 1,\ldots,M$

$$x_{ii}(0) = \min(x_{ii}^- + r_i - d_{ii}, x_i^{MAX}) \tag{4a}$$
$$x_{ij}(0) = x_{ij}^- - d_{ij},\ j \neq i \tag{4b}$$

2. For all $i, j = 1,\ldots M$, set

$$x_{ij}(k+1) = x_{ij}(k) + u_{ij}(k) \tag{5}$$

   where $u_{ij}(k)$ is defined, for all $i, j = 1,\ldots,M$, in (6).
3. $k = k+1$.
4. If the termination condition (see (7)) is attained go to step 5, otherwise and go to step 2.
5. For all $i, j = 1,\ldots M$, set $x_{ij}^+ = x_{ij}(k)$, while the memory to be allocated by the swap area by task T_i is $n_i(k)$, for all $i = 1,\ldots,M$.

A far as the allocation law (5) is concerned, we define u_{ij} as follows.
First, concerning the case $j \neq i$, for all $i = 1,\ldots,M$, two different update rules are defined, depending on the values of $e_j(k)$ and $n_i(k)$ defined in Equations (2) and (3).

- In case $e_j(k) \geq 0$, the update rule is
$$u_{ji} = \lambda \min\left(\frac{\max\{n_i(k),0\}}{|\mathscr{N}_i|}, \frac{e_j(k)}{|\mathscr{N}_j|}\right) \tag{6a}$$
where $\lambda \in (0,1]$ is a tuning knob, while $|\mathscr{N}_i|$ and $|\mathscr{N}_j|$ denote the cardinality of $\mathscr{N}_i$ and $\mathscr{N}_j$, respectively (i.e. the number of elements of set $\mathscr{N}_i$ and $\mathscr{N}_j$). It is worth noting this case includes the following situations: (a) when $e_j(k) = 0$, and (b) $e_j(k) > 0$ and $n_i(k) < 0$: both in (a) and (b) no actions are required (i.e., $u_{ji} = 0$). In particular, the case (a) means that no memory in B_j is available to satisfy external requests, while the case (b) corresponds to the case when no external memory is required by the taskset T_i.
- In case $e_j(k) < 0$, there is a need to deallocate some memory from bank B_j, regardless of the value of $n_i(k)$. Thus, the update aims to deallocate memory from B_j previously allocated by the neighbors of bank j, i.e.,
$$u_{ji} = -\lambda \min\left(\frac{|e_j(k)|}{|N_j|}, x_{ji}(k)\right). \tag{6b}$$

Secondly, concerning the case $j = i$, for all $i = 1,\ldots,M$, we define two alternative choices.

- **Choice 1**:
$$u_{ii}(k) = 0 \tag{6c}$$
In this case, no possibility is given for allocating memory from taskset T_i in bank B_i after the initialization. This is not particularly critical, since the initialization step 1) (see equation (4a)) is "greedy", in the sense that priority is given to local allocation. The only drawback of Choice 1 may lie in the fact that, if neighboring banks B_j, $j \in \mathscr{N}_i$ deallocate memory needed by taskset T_i and space is still available on bank B_i, this memory does not get stored locally, but is bounded to be allocated in the swap area.
- **Choice 2**: this choice has been introduced to overcome the problems arising in the application of Choice 1. Indeed, to limit as much as possible the memory to allocate in the swap area, the following alternative rule can be applied:
$$u_{ii}(k) = \min\{x_i^{MAX} - x_{ii}(k), \sum_{j\in\mathscr{N}_i} |\min\{u_{ji}(k),0\}|\} \tag{6d}$$
The rationale of the previous equation is that, if space in B_i is available to local allocation (i.e., if $x_i^{MAX} - x_{ii}(k) > 0$), we try to locally allocate on B_i the memory that is deallocated by the neighboring banks (i.e., represented by the absolute values of the negative terms u_{ji} computed in (6b)).

A last point concerns the termination condition. Recall that all the computations are performed at the OS level. Therefore, the values of $x_{ij}(k)$, for all $i,j = 1,\ldots,M$, are available to the OS at each time step. Therefore, the termination condition is centralized, and is attained when the algorithm converges to a steady state. This, in practice, is detected at the step k when the following inequality is verified
$$\max_{i,j=1,\ldots,M} |x_{ij}(k) - x_{ij}(k-1)| < \varepsilon \tag{7}$$
where ε is a suitable threshold level.

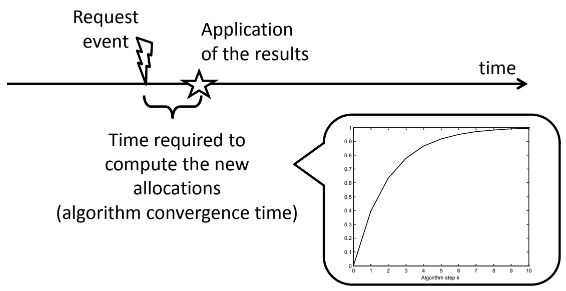

Request event
Application of the results
time
Time required to compute the new allocations (algorithm convergence time)
Algorithm step k

Fig. 2. Event-triggered algorithm

3.3 Application issues

The main issue arising in the application of the proposed scheme is related to the fact that the algorithm is event-based, as sketched in Figure 2. Indeed, after the request reaches the OS, the execution of the RA algorithm is triggered. Information about the status of all the involved processors are gathered by the OS and Algorithm 1 is executed at the OS level. After convergence is achieved, the results are transmitted to the local processors.
By definition, the algorithm is iterative, i.e., it requires that Steps 2.-5. are performed a number of times, until convergence is attained. It is worth noting that each step requires a minimal computational demand (i.e., which basically consists in the computational load required, by the OS, to compute the steering term u_{ij}, for all $i,j = 1,\ldots,M$). Therefore, since the memory requests are expected to arrive with a coarse grain granularity in the order of milliseconds, it is estimated that the memory management algorithm convergence time is negligible with respect to the granularity of the events, which makes the proposed approach practically applicable.

4. SIMULATION RESULTS

In this section we test the proposed RA algorithm considering different multi-processor architectures. In the simulations we will set $\lambda = 0.9$ and $x_i^{MAX} = 100$, for all $i = 1,\ldots,4$. We consider two different experiments.

4.1 Experiment 1

Consider a quad-processor architecture with four tasksets and four memory banks. Figure 3 depicts a scheme of the processor and the connections among memory banks. Neighboring systems are indicated using arrows, and therefore $\mathscr{N}_1 = \{2,4\}$, $\mathscr{N}_2 = \{1,3\}$, $\mathscr{N}_3 = \{2,4\}$, and $\mathscr{N}_4 = \{1,3\}$.
This means each processor can access its local memory bank and two remote memory banks. Figures 4 and 5 show the results of application of RA, related to the cases when Choice 1 (see (6c)) and Choice 2 (see (6d)), respectively, are used for setting $u_{ii}(k)$. The total allocated memory for each taskset is reported in the Figures 4a and 5a where, for each taskset, solid thick and a thin lines represent the total request and the actual allocated memory, respectively. Figures 4b and 5b report the required memory not allocated for each taskset, i.e., the total swapped memory for each taskset at the end of the memory allocation process. Last, the allocation level of each bank is reported in Figures 4c and 5c.

Consider first Figure 4. At instant $t = 20$ each bank is full due to previous local requests. The analysis is focused on two different

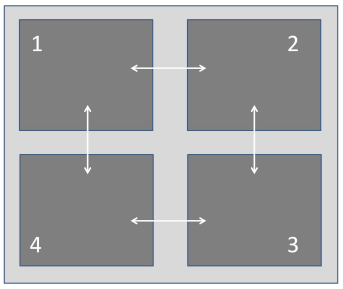
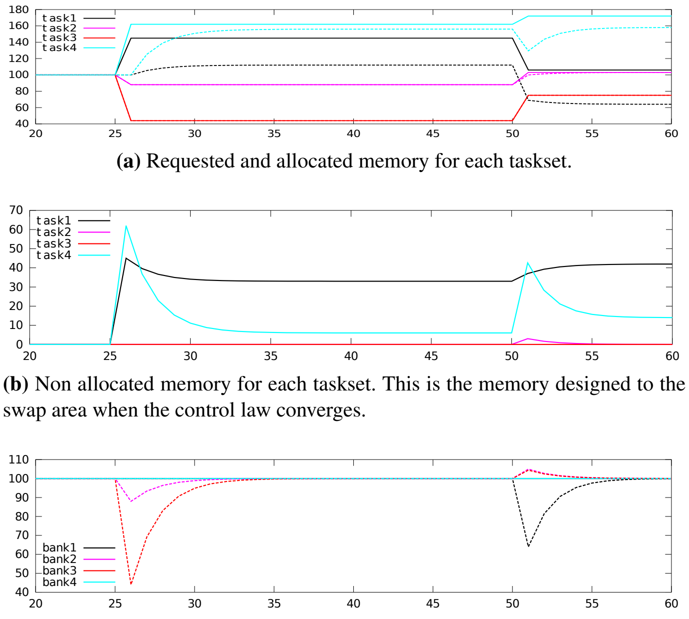


Fig. 3. Sketch of the multiprocessor architecture with four memory banks used in Experiment 1.



(a) Requested and allocated memory for each taskset.

(b) Non allocated memory for each taskset. This is the memory designed to the swap area when the control law converges.

(c) Status of each memory bank.

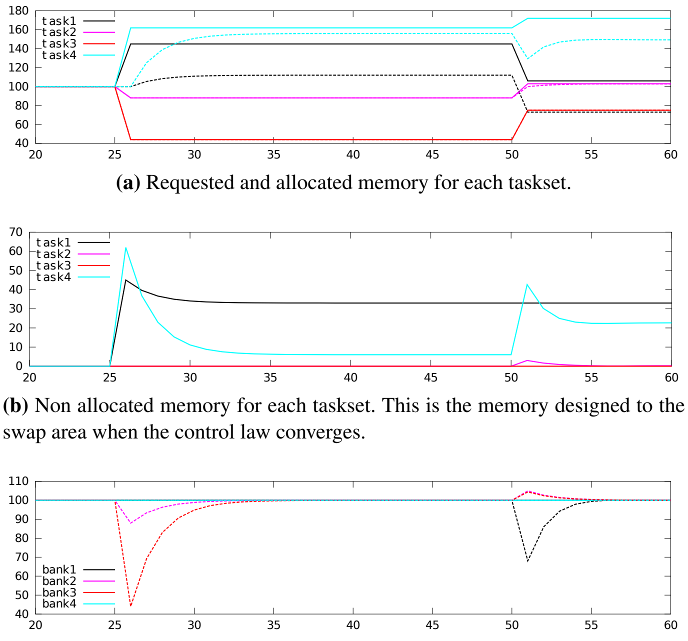
Fig. 4. Simulation results for Experiment 1, Choice 1.

events, occurring at $t = 25$ and $t = 50$. Such events represent the requests for memory allocation and deallocation from the running tasksets and trigger the execution of the proposed RA algorithm. It is worth noticing that the simulation scenario represents a critical case, where the available memory on the local nodes is not sufficient to satisfy the task requests.

At time $t = 25$ the event requests a deallocation of memory for T_2 and T_3 below the maximum bank capacity, which provides free space for external allocation on banks B_2 and B_3. On the other hand, both T_1 and T_4 have a positive net request imposing a remote memory allocation. As a result, T_4 allocates almost all the required memory using the space on bank B_3, while T_1 has a lower external memory allocation on bank B_2. The memory allocation converges at time $t = 33$, and some memory needed by T_1 and T_4 must be allocated in the swap area.

The event at time $t = 50$ allows additional considerations. Since T_3 has a positive net request, it gives priority to local requests and rapidly deallocates memory used by task T_4. After that, the allocation due to the positive allocation request in T_4 is partly satisfied by bank B_1.

Then, another aspect to be considered is the behavior of T_1 starting from time $t = 51$. After the occurrence of the event, the required memory is slightly greater than 100 (see blue line at time $t = 51$ in Figure 4a). However, the swapped memory for task T_1 after the event at time $t = 50$ increases. This behavior is due to the impossibility to recall memory from the swap area. It is worth noticing this has not to be considered as a negative side effect. In fact, the swap-in process of memory from the swap area requires time and energy, and therefore, as discussed, it is not desirable. Moreover, some memory of T_1 is deallocated on bank B_2 to make space for the local memory required by the local T_2, which increased the T_1 memory requests at event $t = 51$. Concerning this point, consider Figure 5, where the results



(a) Requested and allocated memory for each taskset.

(b) Non allocated memory for each taskset. This is the memory designed to the swap area when the control law converges.

(c) Status for each memory bank.

Fig. 5. Simulation results for Experiment 1, Choice 2.

(in the same scenario as in Figure 4) are obtained with equation (6d). It is worth noticing that the only difference between the results in Figure 4 and those reported in Figure 5 arise at $t = 51$ where the swapped memory of T_1 does not increase due to a reclaim on the local bank. As a net consequence, part the swapped memory for T_4 increases, because bank B_1 deallocates some external memory to make space for its local reclaim and T_4 cannot do the same, since its local bank is full of local memory. Remark, however, that in this example the two different policies have equivalent results, in term of total amount of memory to be allocated in the swap area.

Some final comments are in order. First, as expected, the total allocation level for each memory bank is always lower or equal to the maximum available bank space at the end of the transient period (see Figure 4c), i.e., when the resulting allocation levels are transmitted to the local processors to be implemented. Secondly, with both policies (equations (6c) and (6d)) the controller uses all the available memory space on banks to allocate memory, thus minimizing the swapped memory. Last, the allocation of the memory for each application on its own block is satisfied in one step, allocating up to the maximum available memory of the bank. The behavior is justified by the proposed scheme, which gives priority to local allocations over remote ones (see equation (4a)).

4.2 *Experiment 2*

In order to highlight the advantages of the Choice 2 in the selection of $u_{ii}(k)$, consider a two-banks/two-tasksets simulation example, see Figure 6.

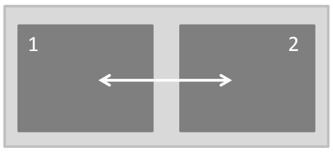
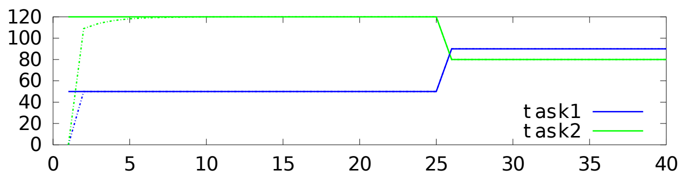
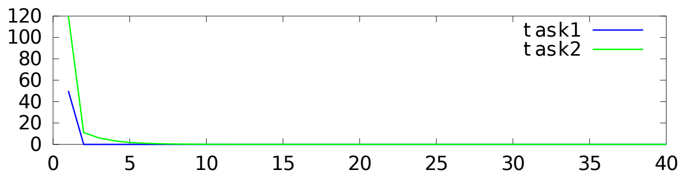


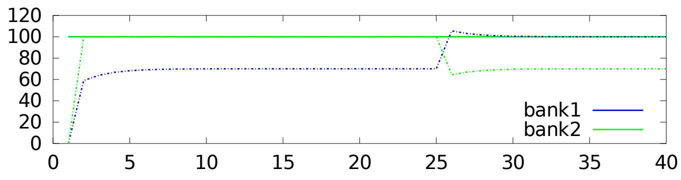
Fig. 6. Sketch of the multiprocessor architecture with two memory banks used in Experiment 2.



(a) Requested and allocated memory for each taskset.



(b) Non allocated memory for each taskset. This is the memory designed to the swap area when the control law converges.



(c) Status for each memory bank.

Fig. 7. Simulation results for Experiment 2.

Figure 7 presents the simulation results. At $t = 0$ no memory is allocated. At $t = 1$, the request of T_2 is greater than the memory available on B_2, and therefore some memory used by T_2 is allocated on bank B_1. However, the event at time $t = 25$ forces bank B_1 to deallocate part of its external memory due to a local request. At the same time, bank B_2 can reclaim such memory portion (provided that Choice 2 is adopted), since T_2 required a deallocation on bank 2 itself. The net result is an empty swap area when the control law reaches the steady state.

5. CONCLUSIONS

The multiprocessor revolution allows to have different processors accessing their local memory controller and the relative RAM portion, while the possibility for each processor to access the remote memory on another processor enables the so-called Non-Uniform-Memory-Access (NUMA) multiprocessor paradigm. The use of the so-called swap area represents a common solution which allows the collectively memory required by the running processes to be greater than the available RAM. In this scenario, the RAM access time and energy depend on the location on both requesting processor and physical memory bank, while the access the swap area is to be always considered the worst solution.

This paper discussed a control-inspired policy to allocate the memory requests by different applications in a NUMA systems, trying to maximize the RAM utilization under the locality requirement. Moreover, the possibility to activate and deactivate the reallocation of deallocated memory from neighboring memory banks represents another feature which pushes the exploration of the memory management in commercial NUMA architectures.
The possibility to allocate the memory which cannot be fit in the local bank in neighboring ones has two great advantages. First, the RAM is better used, since a memory-bound application can *invade* remote memory banks if they are not used, thus reducing the swap overhead. Moreover, the possibility for a memory bank to deallocate the external memory in presence of local memory requests allows to limit the conflicting situations between concurrent applications.

REFERENCES

AMD (2002). Amd hypertransport technology-based system architecture. Technical report, University of Zurich, Department of Informatics.

Blagodurov, S., Zhuravlev, S., Fedorova, A., and Kamali, A. (2010). A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, 557 – 571. ACM, New York, NY, USA.

Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. (2013). Traffic management: a holistic approach to memory placement on numa systems. *SIGPLAN Not.*, 48(4), 381–394. doi:10.1145/2499368.2451157. URL `http://doi.acm.org/10.1145/2499368.2451157`.

Jones, R. (1969). Factors affecting the efficiency of a virtual memory. *Computers, IEEE Transactions on*, C-18(11), 1004–1008. doi:10.1109/T-C.1969.222570.

Majo, Z. and Gross, T.R. (2011). Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *Proceedings of the international symposium on Memory management*, ISMM '11, 11–20. ACM, New York, NY, USA.

Molka, D., Hackenberg, D., Schone, R., and Muller, M. (2009). Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, 261–270. doi:10.1109/PACT.2009.22.

Terraneo, F. and Leva, A. (2013). Feedback-based memory management with active swap-in. In *European Control Conference ECC*.

Ziakas, D., Baum, A., Maddox, R., and Safranek, R. (2010). Intel quickpath interconnect architectural features supporting scalable system architectures. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, 1–6.

Zuo, W., Feng, S., Qi, Z., Weixing, J., Jiaxin, L., Ning, D., Licheng, X., Yuan, T., and Baojun, Q. (2009). Group-caching for noc based multicore cache coherent systems. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, 755–760. doi:10.1109/DATE.2009.5090765.