Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

LBMA and IMAR^2: Weighted lottery based migration strategies for NUMA multiprocessing servers

R. Laso[1] | O. G. Lorenzo[1] | F. F. Rivera[1] | J. C. Cabaleiro[1] | T. F. Pena[1] | J. A. Lorenzo[2]

[1]CiTIUS, Universidade de Santiago de Compostela, Santiago de Compostela, Spain
[2]CY Cergy Paris Université, France

Correspondence
*Laso, Ruben. CiTIUS, Jenaro de la Fuente Domínguez S/N, 15782, Santiago de Compostela, Spain. Email: r.laso@usc.es

Present Address
CiTIUS, Jenaro de la Fuente Domínguez S/N, 15782, Santiago de Compostela, Spain.

Abstract

Multicore NUMA systems present on-board memory hierarchies and communication networks that influence performance when executing shared memory parallel codes. Characterising this influence is complex, and understanding the effect of particular hardware configurations on different codes is of paramount importance. In this paper, monitoring information extracted from hardware counters at runtime is used to characterise the behaviour of each thread for an arbitrary number of multithreaded processes running in a multiprocessing environment. This characterisation is given in terms of number of operations per second, operational intensity, and latency of memory accesses. We propose a runtime tool, executed in user space, that uses this information to guide two different thread migration strategies for improving execution efficiency by increasing locality and affinity without requiring any modification in the running codes. Different configurations of NAS Parallel OpenMP benchmarks running concurrently on multicore NUMA systems were used to validate the benefits of our proposal, in which up to four processes are running simultaneously. In more than the 95% of the executions of our tool, results outperform those of the Operating System (OS) and produces up to 38% improvement in execution time over the OS for heterogeneous workloads, under different and realistic locality and affinity scenarios.

KEYWORDS:
Thread migration; Hardware Counters; Roofline Model; Performance

1 | INTRODUCTION

Current multicore machines feature a diverse set of computing units and on-board memory hierarchies connected by increasingly complex communication networks and protocols. For a parallel code to be correctly and efficiently executed in a multicore system, it must be carefully programmed, paying special attention to expensive memory operations. In addition, the performance of a code also depends on the status of the other processes currently running in the system, in particular, when the code works in a multiprocessing environment. A traditional challenge in high-performance computing (HPC) is to partition application tasks and map them to one of many possible thread-to-core configurations, seeking to achieve the best performance in terms of throughput, delay, power, and resource consumption, among others [1]. The main difficulty lays in the fact that the behaviour of a system can dynamically change when multiple processes are running with several threads each, since each application typically requires different resources that have to be managed efficiently. Furthermore, this demand is variable, given that the number of multithreaded processes can be large and change dynamically in general-purpose systems. Additionally, it must be taken into account the number of mapping and scheduling choices increases exponentially with the number of cores and threads.

Thread scheduling often produces a high impact on performance, since it can affect locality and affinity among threads, cores, and data [2]. Given that, precise and realistic performance models can help to understand the aforementioned impact, and to find better locations for every running thread in the system.

Several models have been proposed to understand the performance of a code running on a given system [3,4,5,6]. In particular, the Roofline Model (RM) [7] is one of the most frequently used, since it offers a balance between simplicity and descriptiveness based just on the number of FLOPS (Floating Point Operations per Second) and the OI (Operational Intensity), defined as the number of FLOPs per byte of DRAM traffic (FLOPs/B). However, in a non-uniform memory access (NUMA) system, the distance and connection to memory cells from different cores may induce large variations in memory latency, so the same process may perform differently depending on where it was scheduled, which clearly affects the behaviour of the whole workload in the system. This effect, not considered in the RM, was included in the 3DyRM [8], which extends the original RM with an additional parameter, memory access latency, measured in number of cycles. 3DyRM shows the dynamic evolution of these parameters through time, and uses the information provided by the hardware counters available in modern processors to obtain its defining metrics dynamically. These three parameters, FLOPS, OI, and latency, identify three important factors that influence the performance of parallel codes when executed in a shared memory multiprocessor and, in particular, in NUMA systems. 3DyRM factors are related to the computing speed given by the FLOPS, the balance between memory accesses and computations given by operational intensity, and the memory access delay given by latency.

In this work, we propose a user space tool that uses the parameters that define the 3DyRM to guide two different strategies for migrating threads in shared memory systems. We particularly focus on multicore NUMA servers, potentially with multiple processes running concurrently. The concept is to use the defining parameters of 3DyRM to find dynamically the aforementioned thread-to-core configuration which gives the best performance. Thus, the problem might be defined as an optimisation problem, optimising 3DyRM parameters, to maximise global performance. The proposed methodologies are iterative methods based on weighted lottery algorithms. Several heuristics, each one with a score assigned, are defined to guide decisions taken. Some of these heuristics use an individual utility function that represents the relative importance of each of the 3DyRM parameters to characterise the performance of each parallel thread in terms of locality and affinity. Finally, it is important to note that the proposed migration tool does not need to modify whatsoever the program under inspection, so any application might take immediate profit of it.

The rest of the paper is structured as follows: Section 2 includes a brief discussion on related work and other proposals that can be found on the literature. Section 3 describes how performance information used by our algorithms is acquired. The proposed migration strategies are introduced in Section 4. Section 5 describes the experimental environment. Section 6 shows a representative case of study, the associated results and their discussion. Finally, Section 7 presents the final conclusions and future work.

2 | RELATED WORK

Thread and memory scheduling and mapping for NUMA architectures is a widely spread topic in literature. Default policies favour load-balancing at the expense of locality and ignore NUMA or manycore access latencies while scheduling. Additionally, cc-NUMA architectures require sophisticated and expensive cache coherence protocols which trigger a significant amount of on- and off-chip traffic in the system. State of the art memory pages placement mechanisms interleave pages evenly across NUMA nodes. However, this approach fails to maximise memory throughput in NUMA systems characterised by asymmetric bandwidths and latencies, and sensitive to memory contention and interconnection congestion effects [9]. Different solutions have been proposed, like modifying the Linux kernel for improving load balancing and thread migration algorithms, changing memory pages placement policies, or profiling the executed program for obtaining the optimal thread placement, among others.

Several works have treated scheduling problem with previous profiling phases to compute optimal thread and/or memory locations. In the paper by Li et al. [10], a methodology for measuring NUMA behaviour and a thread mapping algorithm are proposed for memory and I/O applications. A deep analysis phase is applied to a NUMA-aware implementation of an intense data movement application, BBCP. The authors obtain important performance improvements, mainly up to a 220% bandwidth improvement. DeLoc [11] focuses in finding a process/thread mapping that improves locality and alleviates memory congestion. The proposed methodology is divided in several phases. First, an initial profile of application's communications is done. Second, an optimal process/thread mapping is computed using DeLocMap. Finally, application is launched again with the computed mapping. Similarly, in the work by Popov et al. [12], a thread and memory pages mapping algorithm is proposed. In this work, an initial profiling phase is performed, in the form of codelets, for each application. Once the required information is gathered, the optimal mapping for both threads and memory is computed using those codelets as reference. When this distribution is found, the application is launched. Results show up to a 5x speedup for the NAS OpenMP benchmarks. Nevertheless, this kind of approaches lack of generality, having an important drawback in the requirement of a profiling phase, which prevents dynamic scenarios and situations where several users access the server simultaneously.

Other research works have focused on concrete applications. Drebes et al. [13] propose a dynamic task and data placement algorithm that guarantees that all accesses to task output data target the local memory of the accessing core. They also provide a task placement heuristic to improve the locality of task input data on a best effort basis. The privatisation of task data enhances scalability by eliminating false dependencies and enabling fine-grained dynamic control over data placement. Their algorithms are fully automatic, application-independent as long as it uses task parallelism, and performance-portable across NUMA machines. Placement decisions use information about inter-task data dependencies readily available in the runtime system and placement information from the operating system. As we mentioned before, the paper by Li et al. [10], has focused only

in memory and I/O applications using an analysis phase for the application BBCP with successful results. However, this kind of proposals lack of generality and applicability to any program or kind of workload.

The most typical kind of solution relies on OS kernel modifications to change thread scheduling or memory allocation. Dashti et al.[14] propose Carrefour, a modification of the Linux kernel that targets traffic congestion for NUMA systems, obtaining performance improvements of up to $3.6\times$ relative to the default kernel, as well as significant improvements compared to NUMA-aware patchsets available for Linux, such as AutoNUMA. Carrefour uses hardware counters to measure performance, and, according to the authors, it takes global decisions to migrate memory pages. Several works by Chiang et al.[15,16,17] use several kernel modifications to improve thread allocation, deal with memory congestion and improve locality. They have obtained important improvements in performance with PARSEC 3.0 benchmarks. Also, the most recent work by Gureya et al.[9] proposes a novel page placement mechanism based on asymmetric weight page interleaving that combines an analytical performance model of the target NUMA systems with on-line iterative tuning page distribution for a given memory-intensive application. By only migrating memory pages and entrusting thread migration to the OS, they obtain up to a 66% performance improvement. These solutions have an important drawback in the requirement of kernel modifications, which might not be possible to all users due to permissions restrictions and portability.

Finally, some authors have tried to implement solutions that run in user-space, like Lepers et al.[18] who introduce AsymSched, a user-level tool that automatically migrates threads and memory in runtime in order to achieve the best placement. The authors improve the performance of many single and multiple application workloads by increasing the available bandwidth, specially on non-symmetric NUMA systems. By using hardware counters to obtain performance information during runtime, AsymSched decides every second the best thread and memory location by just focusing on maximising the bandwidth for communicating threads. Also, their experiments are performed in AMD processors, so they cannot measure memory accesses latency directly and they have to use workarounds not required in Intel architectures.

The main features of our proposal are that it is adapted but not restricted to multiprocessing systems, it runs in user space, and it does not require any modification or profiling of the target codes running concurrently, nor the Linux kernel. Standard codes are utilised to validate the proposal, and we considered the most commonly used scheduling and mapping solutions as baseline. Finally, our proposal can be applied to any thread-based parallel application, not only for OpenMP codes.

3 | PERFORMANCE PROFILING

The performance of a program can be characterised with a reduced set of metrics, like the Roofline Model[7] does with FLOPS and operational intensity. FLOPS measures the number of floating-point operations per second. OI measures the traffic between the caches and the main memory, and indirectly, it also measures the efficiency of use of the cache hierarchy, since a better use of the cache means fewer accesses to main memory. Nevertheless, FLOPS and OI often are not enough to characterise the performance of the running threads in NUMA systems, where thread placement has a high influence on memory operations latency. Including average latency of memory accesses improves the accuracy of the Roofline Model in this kind of systems. Thus, the 3DyRM[8] model is used in this paper, as it provides a three-dimensional representation of thread performance on a particular placement.

To retrieve the information required by the 3DyRM, Intel PEBS[19] (Intel Processor Based Samples) and Perfmon[20] are used. PEBS is an advanced sampling feature of Intel Core based processors through which the processor directly records samples from specific hardware counters into a designated memory region. Perfmon provides an interface for making easier to retrieve information from these hardware counters. The use of PEBS as a tool to monitor a program execution was already implemented in the work by Lorenzo *et al.*[21], which shows that they can provide information in runtime about the behaviour of a code with low overhead[22]. They provided as well a straightforward thread migration tool tested with toy examples.

It should be mentioned that PEBS might not be fully precise when measuring floating point operations[23]. Nevertheless, since this kind of operations play a very important role in the evaluation of performance, they cannot be discarded. Although an in-depth analysis of the applications' machine code would give a precise metric, our objective is to obtain this value in runtime, in user space and without code instrumentation, so PEBS is still the best alternative.

Finally, some other APIs like PAPI[24] have been considered but discarded, since they do not provide all the low level information of the samples required by the algorithms introduced in this paper.

4 | MIGRATION ALGORITHMS

In this work, we introduce a runtime tool that gathers information provided from hardware counters in user space, without requiring any modification of the executed code, and performing thread migrations in NUMA systems using the algorithms discussed in the following sections. This runtime tool is freely available. The source code and user manual can be downloaded from https://gitlab.citius.usc.es/ruben.laso/migration.

4.1 | Problem formulation

In this section, some notation and formal concepts are introduced in order to explain the migration algorithms described in sections 4.2 and 4.3.

At any given time, in a server there is a set of p processes running, denoted by $\Pi = \{\pi_1, \dots, \pi_p\}$. Each process π_i consists of h_i threads, named $\theta_{i,j}$, with $1 \leq j \leq h_i$. Let us consider that the server consists of a set of N nodes, such that the k-th node has C_k cores and each core is named $\gamma_{k,m}$, $1 \leq k \leq N$, $1 \leq m \leq C_k$. Every T seconds, a migration algorithm is executed, so a sequence of time intervals named $\tau_1, \tau_2, \dots$ are considered. Migration algorithms, when performed, will choose a set of threads, $\hat{\Theta}$, that are considered for migration.

Let us define several metrics to be considered in the decision making process, represented by the following functions:

- $A(\theta_{i,j}, k, \tau_t)$ is the number of accesses of the thread $\theta_{i,j}$ to the memory banks in node k during time interval τ_t.
- $L(\theta_{i,j}, \gamma_{k,m}, \tau_t)$ is the average latency of the memory accesses performed by thread $\theta_{i,j}$ while running in core $\gamma_{k,m}$ during time interval τ_t.
- $O(\theta_{i,j}, \gamma_{k,m}, \tau_t)$ is the number of operations performed by thread $\theta_{i,j}$ while running in core $\gamma_{k,m}$ during time interval τ_t. Two options are available for O. The first one consists in measuring the number of retired instructions, noted as O_R. The other one, O_F, gathers both retired instructions and floating-point operations. The later should measure performance more precisely as a single vector instruction performs several floating-point operations[1]. Also, note that $O_R \leq O_F$ for any thread, at any core, during any interval.
- $I(\theta_{i,j}, \gamma_{k,m}, \tau_t)$ is the measured operational intensity for thread $\theta_{i,j}$ while running in core $\gamma_{k,m}$ during time interval τ_t.

Using these metrics, a certain amount of tickets (a score), q_i, is granted to each thread being migrated to each core, according to the likelihood of improving its performance. Details of this ticket assignation are given at sections 4.2 and 4.3.

Therefore, a set of possible migrations is defined as $M = \{M_1, M_2, \dots\}$. And each possible migration will be denoted as $M_i = \left[\tilde{\Theta}, \tilde{C}, Q\right]$, being $\tilde{\Theta}$ the list of threads to be migrated, $\tilde{C}$ their destination cores, and Q the summation of their granted tickets. The number of threads to be moved in a migration M_i can be one or two, since both a single migration or an interchange, might be actually performed. In the case of a single migration, we would have $\tilde{\Theta} = [\theta_{i,j}]$ and $\tilde{C} = [\gamma_{k,m}]$, so thread $\theta_{i,j}$ would be migrated to core $\gamma_{k,m}$. In the case of an interchange, $\tilde{\Theta} = [\theta_{i,j}, \theta_{i',j'}]$ and $\tilde{C} = [\gamma_{k,m}, \gamma_{k',m'}]$, so $\theta_{i,j}$ would be migrated to $\gamma_{k,m}$ and $\theta_{i',j'}$ to $\gamma_{k',m'}$.

4.2 | LBMA Algorithm

Our first proposal is a weighted lottery algorithm named LBMA (Lottery-Based Migration Algorithm), in which migrations are driven by a reduced set of heuristic rules.

During time intervals $\tau_1, \tau_2, \dots$, hardware information is gathered. In the LBMA algorithm, only the memory operations are taken into account. At the end of each interval, say τ_t, a random set of n threads $\hat{\Theta} \subseteq \Theta$ is considered for migration. For each thread $\theta_{i,j} \in \hat{\Theta}$, running in the core $\gamma_{k,m}$, almost all possible destinations are considered. Cores in the k-th node are discarded, since it is assumed that every core in a node would have a similar behaviour and the same latency for the same memory accesses.

In this algorithm, only two kinds of tickets are considered. On the one hand, q_1 tickets are granted if the possible destination core $\gamma_{k',m'}$ has not hosted any other thread in τ_t. By default, $q_1 = 2$. On the other hand, q_2 tickets are granted if $\gamma_{k',m'}$ is in the node that meets

$$A(\theta_{i,j}, k', \tau_t) = \max_{r=1,\dots,N} A(\theta_{i,j}, r, \tau_t). \tag{1}$$

That is, if $\gamma_{k',m'}$ belongs to the node in which $\theta_{i,j}$ has performed most of its memory operations (its preferred node), q_2 tickets are given. By default, $q_2 = 4$. Tickets can be accumulated, i.e. if a candidate migration meets the requirements for both q_1 and q_2, a total of $q_1 + q_2$ tickets will be awarded. If another thread, $\theta_{i',j'}$, is also running in core $\gamma_{k',m'}$, a swap is considered instead of a single migration, also assigning the corresponding tickets for $\theta_{i',j'}$ being migrated to $\gamma_{k,m}$.

Once that there is a set of candidate migrations for each $\theta_{i,j} \in \hat{\Theta}$, those to be performed are selected using a weighted lottery process to avoid stacking into local minimums. A random number between 0 and Q, with a uniform distribution, is assigned to each candidate migration, so the migrations with higher scores will likely get a higher random number. Finally, the n migrations with the highest random value are finally performed. Algorithm 1 shows the pseudocode of this migration strategy. It should be noted that this algorithm will take a thread and, with high probability, move it to its preferred node, in which its performance is expected to be the best possible if memory access patterns do not change.

[1] Notice that PEBS does not allow checking the vector instructions with integer operands.

Algorithm 1 LBMA migration strategy.

Input: Set of processes $\Pi = \{\pi_1, \pi_2, \ldots, \pi_p\}$.
Set of threads $\Theta = \{\theta_{i,j},\ i = 1, \ldots, p,\ j = 1, \ldots, h_i\}$.
Set of cores $\Gamma = \{\gamma_{k,m},\ k = 1, \ldots, N,\ m = 1, \ldots, C_k\}$.
Number of threads to be migrated n.
Output: Set of migrations to be performed $M = \{M_1, M_2, \ldots, M_n\}$.

```
procedure Migr(Π, Θ, N, Γ, m)
  Θ̂ := {θ_{i,j} | θ_{i,j} ∈ Θ}                              ▷ Set of n threads randomly selected.
  M̂ = ∅                                                      ▷ Candidate migrations is an empty set at the beginning.
  for each θ_{i,j} ∈ Θ̂ do                                    ▷ Compute candidate migrations.
    γ_{k,j} := core hosting θ_{i,j}
    for each γ_{k',j'} ∈ Γ | k' ≠ k do                       ▷ For each core in a different node, search for candidate migrations.
      if γ_{k',j'} is free then
        Q = q_1
        if A(θ_{i,j}, k', τ_t) = max_{r=1,...,N} A(θ_{i,j}, r, τ_t). then
          Q = Q + q_2
        end if
        M̂ = M̂ ∪ {[θ_{i,j}, γ_{k'm'}, Q]}                     ▷ Single migration of θ_{i,j} to γ_{k,m}.
      else
        for each θ_{i',j'} running in C_k do
          Q = 0
          if A(θ_{i,j}, k', τ_t) = max_{r=1,...,N} A(θ_{i,j}, r, τ_t). then
            Q = Q + q_2
          end if
          if A(θ_{i',j'}, k, τ_t) = max_{r=1,...,N} A(θ_{i',j'}, r, τ_t). then
            Q = Q + q_2
          end if
          M̂ = M̂ ∪ {[θ_{i,j}, θ_{i',j'}, γ_{k,m}, γ_{k',m'}, Q]}  ▷ Interchange, θ_{i,j} would switch to γ_{k,m}, and θ_{i',j'} to γ_{k',m'}.
        end for
      end if
    end for
  end for
  M := Weighted_Lottery_Selection(M̂, n)                      ▷ Perform weighted selection process of candidates M̂.
  return M
end procedure
```

4.3 | IMAR^2 algorithm

As a refinement of the algorithm discussed in Section 4.2 (extending previous work by Lorenzo *et al.*[25]), an algorithm named Interchange and Migration Algorithm with Performance Record and Rollback (IMAR^2) is introduced.

The algorithm is executed every T seconds, being the value of T variable within the interval $[T_{min}, T_{max}]$ according to the system global performance as explained later. IMAR^2 uses the information provided by the 3DyRM to guide the decision process, such that for a given thread $\theta_{i,j}$, its performance while running in the core $\gamma_{k,m}$ can be numerically described by the function $P(\theta_{i,j}, \gamma_{k,m}, \tau_t)$. Further details of this function will be given in Section 4.3.1.

Using function P, the selection of threads in $\hat{\Theta}$ can be improved to select those with worst performance. The average performance of process π_i in the time interval τ_t will be

$$\bar{P}(\pi_i, \tau_t) = \frac{1}{h_i} \sum_{j=1}^{h_i} P(\theta_{i,j}, \gamma_{k,m}, \tau_t), \tag{2}$$

where $\gamma_{k,m}$ is the core in which the thread $\theta_{i,j}$ was running during that time interval.

So, $\hat{\Theta}$ will be formed by the n threads with the lowest relative performance,

$$\hat{P}(\theta_{i,j}, \gamma_{k,m}, \tau_t) = \frac{P(\theta_{i,j}, \gamma_{k,m}, \tau_t)}{\bar{P}(\pi_i, \tau_t)}, \tag{3}$$

and meeting that it is under a given threshold δ_r, $0 < \delta_r < 1$. So, $\forall\, \theta_{i,j} \in \hat{\Theta}$, $\hat{P}(\theta_{i,j}, \gamma_{k,m}, \tau_t) < \delta_r$. Using relative performance per process prevents unfair selections of threads in $\hat{\Theta}$. Imagine two processes π_a and π_b, being π_a much more computationally intensive. Threads of π_b will likely have lower values of performance, so only those would be considered in $\hat{\Theta}$. As a consequence, threads of π_a process would never be migrated, and their performance would never be improved. Also, a minimum threshold will prevent unnecessary migrations. Let us take a scenario in which the worst thread has a relative performance of 0.98. In that situation, it is not appropriate to say that there is a thread with bad performance as all threads are performing almost equally so, probably, its mapping is already optimal.

Candidate migrations are considered for each $\theta_{i,j} \in \hat{\Theta}$ in a similar way as done Section 4.2, but considering now four situations:

- q_1 tickets are granted if destination core $\gamma_{k',m'}$ was not hosting threads during τ_t. By default, $q_1 = 2$.
- q_2 tickets are assigned to the cores in the preferred node. By default, $q_2 = 4$.
- q_3 tickets are given according to the previous performance of $\theta_{i,j}$ in the considered destination core $\gamma_{k',m'}$. Performance during τ_t is compared with the last performance measure obtained by $\theta_{i,j}$ when running in a core in node k'. If the previous performance was better, $q_3 = 4$. If it was worse, $q_3 = 1$. Otherwise, $q_3 = 2$.
- q_4 tickets are given if a swap is considered between $\theta_{i,j}$ and $\theta_{i',j'}$, and $\hat{P}(\theta_{i',j'}, \gamma_{k',m'}, \tau_t) < \delta_r$. By default, $q_4 = 3$.

Once that there are candidate migrations for every $\hat{\theta}_{i,j} \in \hat{\Theta}$, a weighted lottery process is used to select those to be performed. Algorithm 2 shows the pseudocode of migration selection process of IMAR^2.

Finally, after τ_{t+1}, global performance is computed using

$$P_g(\tau_{t+1}) = \sum_{i=1}^{P} \sum_{j=1}^{h_i} P(\theta_{i,j}, \gamma_{k,m}, \tau_{t+1}), \tag{4}$$

where $\gamma_{k,m}$ is the core in which $\theta_{i,j}$ was running during interval τ_{t+1}, and it is compared to $P_g(\tau_t)$. Three situations are considered:

- If $P_g(\tau_{t+1}) > P_g(\tau_t)$, performance has improved and migrations are considered successful, so a new time interval is selected as $T = \max(T/2, T_{min})$, and migration process is accelerated (migrations will be performed more often).
- If $P_g(\tau_{t+1}) < \delta_g P_g(\tau_t)$, $0 < \delta_g < 1$ migrations implied a heavy loss of performance, so a rollback is performed (migrations are undone) and migration process is decelerated making $T = \min(2T, T_{max})$.
- Otherwise, migrations are accepted and T keeps its previous value.

4.3.1 | Performance function P

Several definitions can be considered for the aforementioned function $P(\theta_{i,j}, \gamma_{k,m}, \tau_t)$. Depending on which characteristic we are looking to optimise, trivial definitions would include:

$$P(\theta_{i,j}, \gamma_{k,m}, \tau_t) := L(\theta_{i,j}, \gamma_{k,m}, \tau_t), \tag{5}$$

$$P(\theta_{i,j}, \gamma_{k,m}, \tau_t) := O(\theta_{i,j}, \gamma_{k,m}, \tau_t), \tag{6}$$

$$P(\theta_{i,j}, \gamma_{k,m}, \tau_t) := I(\theta_{i,j}, \gamma_{k,m}, \tau_t). \tag{7}$$

Equations (5) to (7) would make of IMAR^2 a single-objective optimisation algorithm[26,27], as only one parameter of 3DyRM is intended to be optimised. Even though, these three elements should be improved to increase performance in a NUMA server, resulting in a multi-objective optimisation problem. To handle this, a scalarisation[28] is proposed,

$$P(\theta_{i,j}, \gamma_{k,m}, \tau_t) := \frac{O \times I}{L}, \tag{8}$$

and the problem is turned back into a single-objective optimisation problem. This equation is inspired in the weighted product and product of objectives strategy for multi-objective optimisation[27] and the Keeney-Raiffa utility function[29].

Algorithm 2 IMAR^2 migration strategy.

Input: Set of processes $\Pi = \{\pi_1, \pi_2, \ldots, \pi_p\}$.
Set of threads $\Theta = \left\{\Theta_{i,j},\ i = 1, \ldots, p,\ j = 1, \ldots, h_i\right\}$.
Set of cores $\Gamma = \left\{\gamma_{k,m},\ k = 1, \ldots, N,\ m = 1, \ldots, C_k\right\}$.
Relative performance threshold δ_r.
Number of threads to be migrated n.
Output: Set of migrations to be performed $M = \{M_1, M_2, \ldots, M_n\}$.

procedure IMAR^2($\Pi, \Theta, \Gamma, \delta_r, m$)
  $\hat{\Theta} = \left\{\theta_{i,j} \in \Theta \mid P(\theta_j, N_j)/\bar{P}(\pi_i, \tau_t) < \delta_r\right\}$ ▷ Select n threads with worst relative performance and under $\hat{P} < \delta_r$.
  $\hat{M} = \varnothing$ ▷ Candidate migrations is an empty set at the beginning.
  for each $\theta_{i,j} \in \hat{\Theta}$ **do** ▷ Compute candidate migrations.
    $\gamma_{k,m} :=$ core hosting $\theta_{i,j}$
    for each $\gamma_{k',m'} \in \Gamma \mid k' \neq k$ **do** ▷ For each core in a different node, search for candidate migrations.
      if C_k is free **then**
        $Q = q_1 + \text{Tickets}\left(\hat{\theta}_{i,j}, \gamma_{k',m'}\right)$ ▷ Compute tickets for migration of $\theta_{i,j}$ to $\gamma_{k',m'}$.
        $\hat{M} = \hat{M} \cup \left\{\left[\theta_{i,j}, \gamma_{k',m'}, Q\right]\right\}$ ▷ Single migration of $\theta_{i,j}$ to $\gamma_{k',m'}$.
      else
        for each $\theta_{i',j'}$ running in $\gamma_{k',m'}$ **do**
          $Q = \text{Tickets}\left(\theta_{i,j}, \gamma_{k',m'}\right) + \text{Tickets}\left(\theta_{i',j'}, \gamma_{k',m'}\right)$
          $\hat{M} = \hat{M} \cup \left\{\left[\theta_{i,j}, \theta_{i',j'}, \gamma_{k',m'}, \gamma_{k,m}, Q\right]\right\}$ ▷ Swap, $\theta_{i,j}$ would moved to $\gamma_{k',m'}$, and $\theta_{i',j'}$ to $\gamma_{k,m}$.
        end for
      end if
    end for
  end for
  $M :=$ Weighted_Lottery_Selection $\left(\hat{M}, n\right)$ ▷ Perform weighted selection process of candidates $\hat{M}$.
  return M
end procedure

5 | EXPERIMENTAL ENVIRONMENT

Experiments included in this work have been run in two different servers, whose topologies are shown in Figure 1 and include:

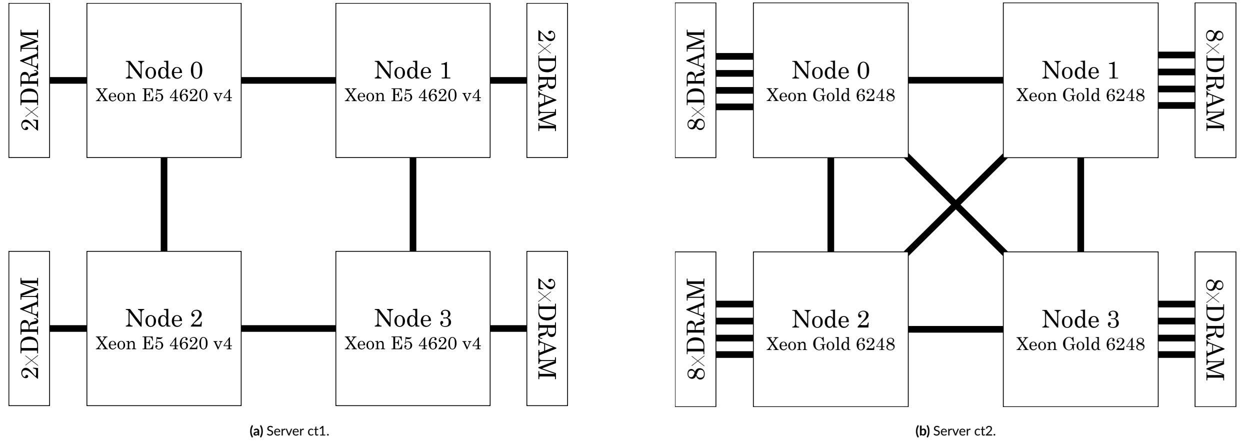
- Server ct1 (Figure 1a): A Debian GNU/Linux 9, kernel version 5.1.15 composed of four nodes with Intel Xeon E5-4620 v4 with 10 cores each (40 in total), Broadwell-EP architecture, 25 MB L3 cache, 2.1 GHz-2.6 GHz, and 128 GB of RAM. Only one of four available memory channels is used in this server for each node.

- Server ct2 (Figure 1b): A Debian GNU/Linux 9, kernel version 4.18.0 composed of four nodes with Intel Xeon Gold 6248 with 20 cores each (80 in total), Cascade Lake architecture, 27.5 MB L3 cache, 2.50GHz-3.9GHz, and 1 TB of RAM. All memory channels are in use and all memories are interconnected among themselves, so 2-hop accesses are in reality 1-hop accesses with a slightly higher latency, which will be named crossed accesses from now on.

Table 1 shows the results of the `numademo` [30] with the STREAM [31] benchmarks. These benchmarks are useful to have an approximated idea of the differences between local and remote accesses in both systems. It can be noticed that Server ct1 penalises remote accesses more, especially when two hops are required. In Server ct2, crossed accesses are slightly slower than one hop accesses. According to these results, we can state that data and thread placement are much more critical in Server ct1 than ct2.

6 | EXPERIMENTAL RESULTS

This section includes and details the set of experiments performed to validate our proposal and a discussion of the obtained results.



(a) Server ct1. **(b)** Server ct2.

FIGURE 1 Network topologies of Server ct1 and Server ct2.

TABLE 1 Sustained bandwidth in MB/s, and difference with respect to local accesses, for benchmarks of `numademo`.

		No policy	Local	Interleaved	1-hop	2-hop / Crossed
Copy	ct1	10400.67 (-0.08%)	10408.79	7827.77 (-24.80%)	7430.09 (-28.62%)	6175.35 (-40.67%)
	ct2	11235.04 (-0.36%)	11275.83	8993.08 (-20.24%)	8403.35 (-25.47%)	8217.85 (-27.12%)
Scale	ct1	10393.88 (+0.01%)	10393.14	7886.04 (-24.12%)	7503.10 (-27.81%)	6228.82 (-40.07%)
	ct2	10995.56 (-0.22%)	11019.86	8748.67 (-20.61%)	8519.58 (-22.69%)	8392.77 (-23.84%)
Add	ct1	11227.75 (-0.12%)	11240.73	7021.77 (-37.53%)	6408.94 (-42.98%)	5987.48 (-46.73%)
	ct2	12432.71 (+0.03%)	12429.20	8381.38 (-32.57%)	7459.52 (-39.98%)	7350.17 (-40.86%)
Triad	ct1	11287.11 (-0.00%)	11287.58	6991.46 (-38.06%)	6387.35 (-43.41%)	5954.96 (-47.24%)
	ct2	12059.22 (-0.05%)	12065.84	8185.82 (-32.16%)	7421.36 (-38.49%)	7245.16 (-39.95%)

In our experimental methodology, we intended to simulate a situation where several users launch their applications in a server, which would have to handle a dynamic workload, taking as reference the experiments in the Lepers et al. work [18]. The idea behind this setup is to have, in a given time, several kinds of programs running in the system, as well as idle users. In this scenario, thread placement in a NUMA server is complex and the OS might struggle handling it.

The NPB-OMP benchmark suite [32] was used to test our thread migration proposal, because these benchmarks are broadly used and their diverse behaviour when executed is well known. They are used in many related works. Particularly, eight benchmarks have been considered: BT (Block Tri-diagonal solver), CG (Conjugate gradient, irregular memory access and communication), DC (Data Cube, unstructured computation), EP (Embarrassingly parallel), IS (Integer sort), LU (Lower-Upper Gauss-Seidel solver), SP (Scalar Penta-diagonal solver), UA (Unstructured Adaptive mesh, dynamic and irregular memory access). Size C of all benchmarks has been used, except for DC, which has size B.

Our tests consisted in simulating four users concurrently accessing the system. In order to have a use case that covers many different situations, we combined the benchmarks among the users in such a way that there are moments in which the four users execute different codes, other moments in which two or more codes are concurrently executed by several users, and moments in which the number of idle users change dynamically and in a parametrised way. Therefore, each user launches the eight applications in different order, with some fixed idle time between the end of an application and the launch of the next one. Idle times are selected so each user will spent, approximately, 100%, 80%, 60% or 40% of time with real computations. With this approach we try to simulate a scenario with real users, in which different codes are running in the system at any moment, being the workload not constant through time. So, the “fictional” users will run:

- User 1: BT, CG, DC, EP, IS, LU, SP, UA.
- User 2: DC, EP, IS, LU, SP, UA, BT, CG.
- User 3: IS, LU, SP, UA, BT, CG, DC, EP.

- User 4: SP, UA, BT, CG, DC, EP, IS, LU.

A graphical representation of an example of how the tests are performed is shown in Figure 2. Every process in the benchmarks is executed

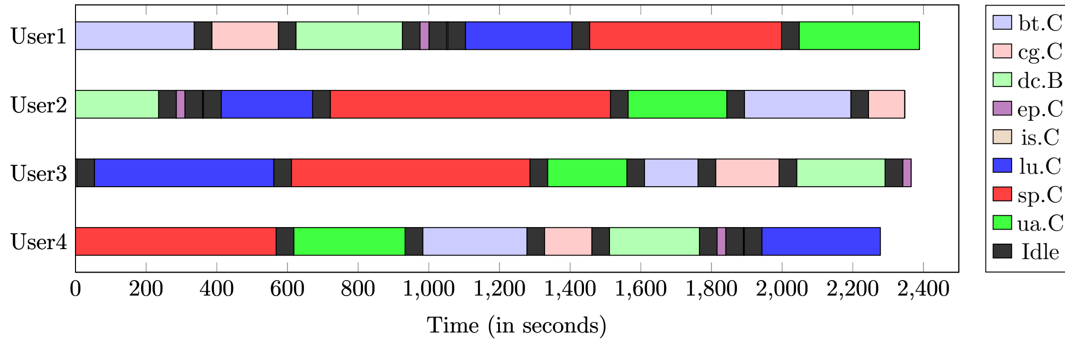


FIGURE 2 Benchmarks execution example.

in the same number of cores, that is 10 cores per user in Server ct1, and 20 in Sever ct2. Consequently, each application will use just 10 and 20 threads for Server ct1 and Server ct2, respectively.

Different conditions and configurations have been considered in the execution of the benchmarks, which will be noted as follows. These configurations are broadly used in the literature as references for validation purposes.

- Baseline: Thread mapping and memory placement is fully under control of the operating system.
- Direct: Each user is granted a memory node, so their processes are directly mapped using the `numactl` command [30]. This way, each benchmark will allocate its threads running in the same node as its data, as long as the memory cell is large enough. This is a common option used by experienced users who know the limits and behaviour of their parallel applications [33,34].
- Interleave: Memory pages are distributed evenly across NUMA nodes using the `numactl` utility. Thread migration is still under responsibility of the OS. Like direct mapping, this is another popular option when executing programs in NUMA servers.
- LBMA: Thread mapping controlled by the algorithm shown in Section 4.2, with $n = 1$ migrations per iteration and $T = 1$, so a migration is performed every second. The initial thread mapping is set by the OS.
- IMAR^2: Thread mapping is controlled by the algorithm described in Section 4.3, using equation (8) and $O = O_F$. Again, the initial thread mapping is set by the OS. By default, $\delta_r = 0.8$, $\delta_g = 0.9$, $n = 1$, $T = 1$ second, $T_{min} = 1$ second, $T_{max} = 4$ seconds.
- IMAR^2_L: version of IMAR^2 that only uses average memory latency for performance evaluation, according to equation (5).
- IMAR$^2_{O_F}$: version of IMAR^2 that only uses executed operations, O_F, to quantify performance, as shown in equation (6).
- IMAR^2_I: version of IMAR^2 that only uses operational intensity to measure performance, check equation (7).
- IMAR$^2_{O_R,L,I}$: version of IMAR^2 where performance is measured with equation (8) and $O = O_R$.
- IMAR$^2_{O_R}$: version of IMAR^2 where equation (6) is used for measuring performance, and $O = O_R$.

Values of the parameters of LBMA and the different configurations of IMAR^2 have been selected after several experiments, non included in this paper, for finding those which fit well in several servers.

6.1 | Server ct1

All results shown in this section reflect the average of 10 different executions of our simulated four-users scenario. Results obtained in Server ct1 are shown in Tables 2 and 3 and Figures 3 and 4. Table 2 shows the wall time of the simulated workload, that is, the time it takes the slowest user to complete all its individual benchmarks plus the time spent idle. Table 3 shows the time spent just executing computation, as given by the NAS

benchmarks; this time is more precise than the wall time in Table 2 and will be used for comparison. Note that the effective computation time decreases as the idle time rises; when a user is idle more resources are available to the others, so the applications may run faster. Figure 3 shows the effective computation time of Table 3 normalised to the OS time for easier comparison. Figure 4 shows the mean effective computation time of all the executions of each particular benchmark, including all the users; this figure allows to compare the effects of each migration strategy on each particular benchmark.

TABLE 2 Execution times, including idle periods, and standard error in seconds of benchmarks in Server ct1. Best time for each row is shown in bold.

Idle time	Baseline	Direct	Interleave	LBMA	IMAR^2	IMAR^2_L	IMAR$^2_{O_F}$	IMAR^2_I	IMAR$^2_{O_R,L,I}$	IMAR$^2_{O_R}$
0	2365 ± 62	2390 ± 14	2417 ± 32	2026 ± 46	1993 ± 49	2186 ± 52	2002 ± 37	**1969** ± 60	1982 ± 36	2075 ± 63
50	2516 ± 56	2823 ± 39	2479 ± 29	**1983** ± 43	2151 ± 42	2199 ± 42	2182 ± 40	2104 ± 52	2088 ± 37	2139 ± 42
100	2849 ± 50	2899 ± 37	2756 ± 46	**2226** ± 24	2307 ± 29	2449 ± 49	2398 ± 44	2489 ± 63	2334 ± 51	2457 ± 46
200	3413 ± 24	3362 ± 31	3425 ± 53	**2863** ± 24	2957 ± 34	3018 ± 36	3054 ± 25	3083 ± 32	3006 ± 25	3023 ± 33

TABLE 3 Effective computation times and standard error in seconds of benchmarks in Server ct1. Best time for each row is noted in bold.

Idle time	Baseline	Direct	Interleave	LBMA	IMAR^2	IMAR^2_L	IMAR$^2_{O_F}$	IMAR^2_I	IMAR$^2_{O_R,L,I}$	IMAR$^2_{O_R}$
0	2158 ± 37	2311 ± 39	2233 ± 35	1719 ± 26	1658 ± 36	1867 ± 44	1688 ± 16	1734 ± 38	1744 ± 21	**1642** ± 31
50	1997 ± 39	2225 ± 39	1971 ± 26	**1384** ± 16	1490 ± 16	1521 ± 22	1529 ± 32	1532 ± 33	1483 ± 18	1476 ± 27
100	1877 ± 39	1936 ± 14	1861 ± 48	1384 ± 16	**1309** ± 16	1379 ± 24	1376 ± 19	1468 ± 32	1359 ± 16	1386 ± 20
200	1700 ± 39	1674 ± 22	1647 ± 36	**1053** ± 19	1177 ± 23	1253 ± 32	1284 ± 24	1310 ± 16	1225 ± 20	1273 ± 24

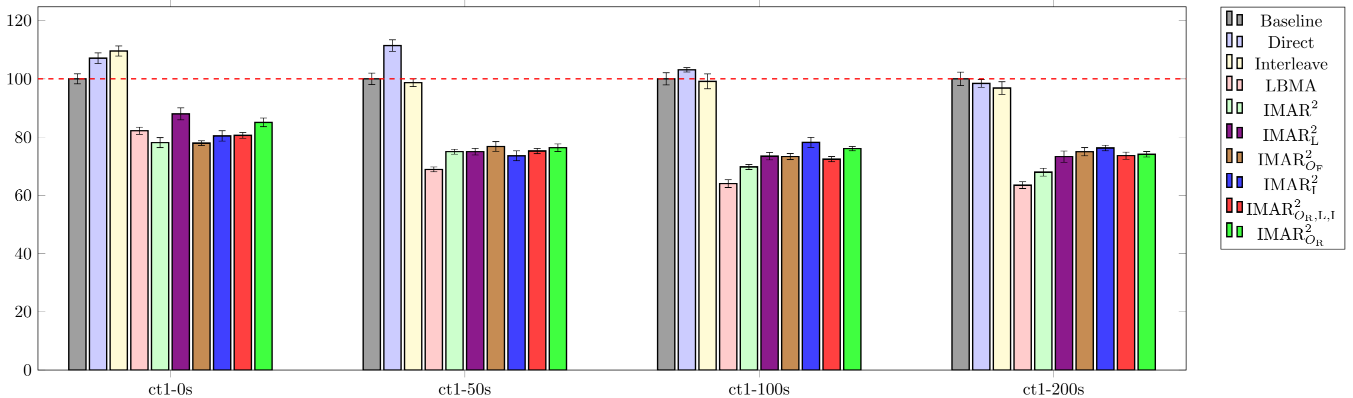


FIGURE 3 Normalised execution times of the benchmarks in Server ct1. Only computation time is considered.

Even though direct mapping is theoretically the best choice, actual results show that it is not necessarily true for every server and every workload, as shown in Figure 3. In fact, direct mapping obtains similar execution times than OS. Also, interleaved mapping, another popular option in NUMA servers, matches OS performance.

Generally, LBMA and the default version of IMAR^2 produce the best results. Both, execution times and standard deviation, are highly improved in this server using migrations algorithms. That is, benchmarks typically run between 20% and 30% (up to a 38%) faster and execution times are more stable between different executions, see Table 3. With larger idle times, results are better, because both LBMA and IMAR^2 have more time to improve thread location in the system, optimising resource usage. IMAR$^2_{O_R}$ gets great results with high occupancy and worsens as the user idle time rises, this is due to the fact that instruction count is a more precise measurement when the system is stressed, but not so relevant when it is not.

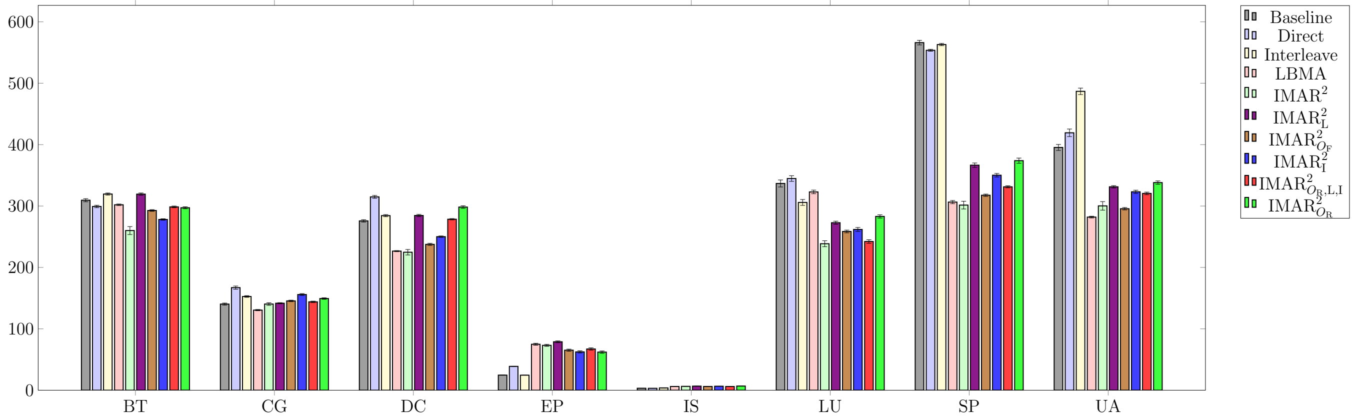


FIGURE 4 Individual benchmarks execution times, in seconds, in Server ct1.

In the case of LBMA, results show that migrating random threads with simple heuristics is a good enough alternative in servers with a high NUMA behaviour. Also, avoiding rollback helps to keep performance as this operation has a high overhead. However, some guidance based on performance metrics is required to obtain more stable and predictable results so, in the authors' opinion, IMAR^2 is still the preferred alternative.

Finally, it is interesting to compare results of default IMAR^2 with IMAR^2_L, IMAR$^2_{O_F}$ and IMAR^2_I. Using just one of the metrics of 3DyRM is not enough to characterise the performance of a thread in NUMA systems, so IMAR^2, using equation (8), gets better results as shown in Figure 3. Also, IMAR$^2_{O_R,L,I}$, based on retired instructions, obtains worse results than the default IMAR^2, which uses the executed operations. This is mainly due to vector instructions, as a single instruction execute multiple operations. For vectorised programs, which are common in HPC, just measuring retired instructions is a bad performance indicator. Additionally, vector instructions are very sensitive to latency, as more data should be loaded from memory, reinforcing their relevance.

When analysing individual benchmarks performance (see Figure 4), migration strategies manage to improve execution times of all benchmarks but EP and IS. Note that these benchmarks have the lowest execution times, around 24 and 3 seconds, respectively. With so little time to perform migrations, the initial location of threads (chosen by OS) plays a very significant role and there is no time for improvements. Consequently, a poor initial location causes a very low performance. For the rest of benchmarks, with higher execution times, migration algorithms succeed in compensating the possible initial location of threads. Especially for SP benchmark, execution times are improved up to a 43%, which is the benchmark with highest execution times. The SP benchmark is heavily memory bound, and requires a large memory bandwidth, so the fact that it improves with the migration algorithms shows that they are working as intended. As conclusion, these results show that throughput is highly improved when using LBMA or IMAR^2.

6.2 | Server ct2

The results obtained in Server ct2 are shown in Tables 4, and 5 and Figures 5 and 6. These tables and figures are analogous as the ones for Server ct1, explained in subsection 6.1.

TABLE 4 Execution times, including idle periods, and standard error in seconds of benchmarks in Server ct2. Best time for each row is shown in bold.

Idle time	Baseline	Direct	Interleave	LBMA	IMAR^2	IMAR^2_L	IMAR$^2_{O_F}$	IMAR^2_I	IMAR$^2_{O_R,L,I}$	IMAR$^2_{O_R}$
0	389 ± 3.9	368 ± 3.1	600 ± 4.8	375 ± 2.0	365 ± 3.0	371 ± 6.7	362 ± 3.3	$\mathbf{360} \pm 3.3$	363 ± 2.3	368 ± 10
12	457 ± 7.0	443 ± 4.6	565 ± 3.1	440 ± 1.6	435 ± 3.5	439 ± 2.8	429 ± 3.2	435 ± 3.3	$\mathbf{432} \pm 2.9$	433 ± 2.8
24	$\mathbf{526} \pm 3.5$	543 ± 4.4	605 ± 2.7	548 ± 4.6	549 ± 7.8	538 ± 5.1	541 ± 5.8	534 ± 6.2	539 ± 4.6	530 ± 4.5
48	$\mathbf{700} \pm 3.6$	747 ± 2.2	764 ± 5.1	736 ± 4.7	737 ± 5.2	728 ± 5.8	740 ± 5.6	732 ± 9.4	732 ± 6.5	723 ± 5.6

In Server ct2, LBMA and IMAR^2 still improve execution times compared to OS in most cases. Nevertheless, improvement is lower (between 3% and 14%) and, in the worst scenario, execution times have increased 1.77% compared to OS, as shown in Figure 5. These are expected results, as the NUMA effects in Server ct2 are not so significant as in ct1 and, consequently, margin for improvement with migrations is much smaller. Even

TABLE 5 Effective computation times and standard error in seconds of benchmarks in Server ct2. Best time for each row is in bold.

Idle time	Baseline	Direct	Interleave	LBMA	IMAR^2	IMAR^2_L	IMAR$^2_{\text{O}_\text{F}}$	IMAR^2_I	IMAR$^2_{\text{O}_\text{R},\text{L},\text{I}}$	IMAR$^2_{\text{O}_\text{R}}$
0	374 ± 1.6	**303** ± 0.3	583 ± 3.9	329 ± 3.4	330 ± 1.7	328 ± 3.0	324 ± 3.0	340 ± 2.4	326 ± 1.8	324 ± 1.7
12	341 ± 6.0	303 ± 0.4	446 ± 2.3	310 ± 1.5	314 ± 2.4	305 ± 3.5	302 ± 1.7	315 ± 2.0	306 ± 2.1	**301** ± 1.6
24	317 ± 2.5	304 ± 0.4	398 ± 2.4	304 ± 3.8	311 ± 2.8	299 ± 3.6	**295** ± 3.8	309 ± 4.3	306 ± 3.7	298 ± 2.3
48	388 ± 2.9	302 ± 0.1	398 ± 2.4	280 ± 4.0	293 ± 3.4	**279** ± 3.6	286 ± 4.9	290 ± 2.6	282 ± 2.2	282 ± 5.4

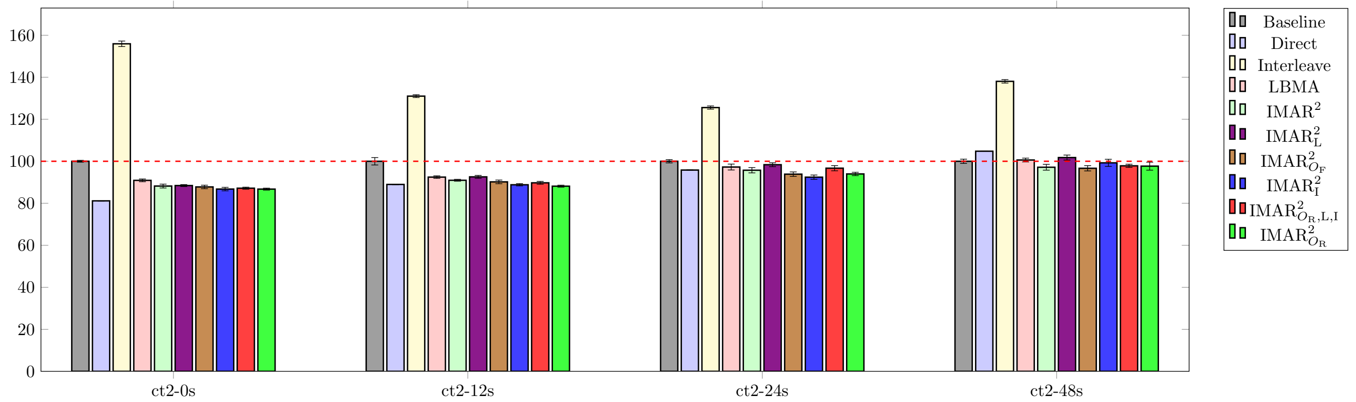


FIGURE 5 Normalised execution times of the benchmarks in Server ct2. Only computation time is considered.

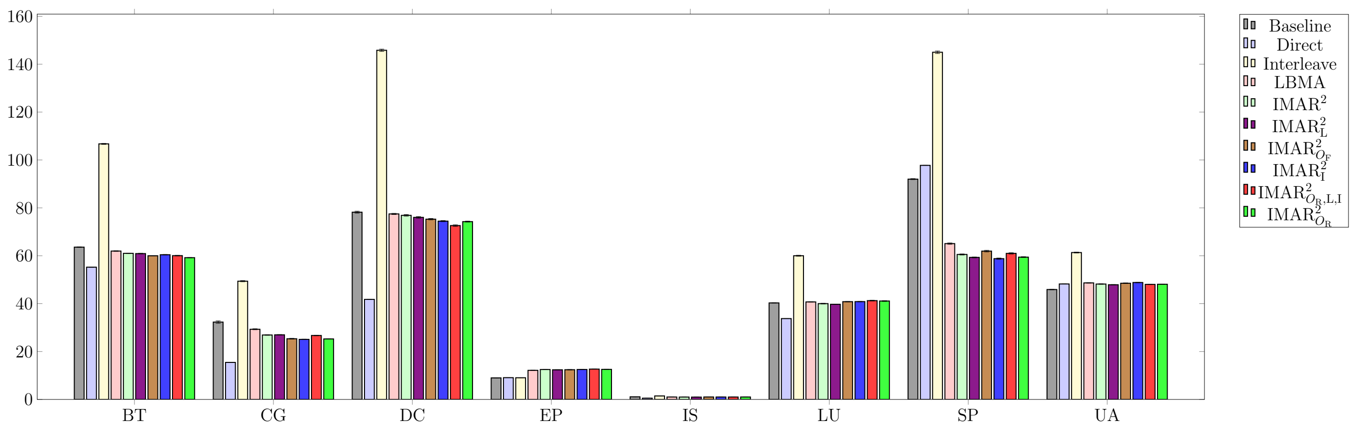


FIGURE 6 Individual benchmarks execution times, in seconds, in Server ct2.

though LBMA still obtains good results, the lack of use of the complete 3DyRM information makes it perform worse than IMAR^2 alternatives. Considering IMAR^2 variations, it can be noticed that IMAR^2_L (only using latency) is the only IMAR^2 variation that increase execution times compared to OS placement when idle time reaches 48 s (see Figure 5). IMAR$^2_{\text{O}_\text{F}}$ achieves great results as it is only concerned about executed operations, much more relevant in this server. Also, it should be noted that default IMAR^2, using all the 3DyRM information, is the algorithm that achieves good results more consistently and, generally, close to the best ones, as shown in Table 4. Finally, interleaved mapping gets the worst results due to the high traffic in the interconnection network, that gets congested, affecting performance severely.

Observing individual benchmarks results, direct mapping usually obtain the best results. Note that all migration strategies depend on the OS for memory mapping, so if a direct memory mapping, where each user has a different memory node for herself, is optimal, our migration strategies may never reach the best time. Adding pages or memory migration to the algorithms may be needed in these cases. Nevertheless, LBMA and IMAR^2 usually obtain slightly better execution times than OS. Again, SP benchmark is the one that improves the most using migration algorithms. EP still does not improve its results, but, as its execution times are very low, its impact on global throughput is reduced.

7 | CONCLUSIONS AND FUTURE WORK

In this work, two new thread migration algorithms for NUMA servers, named LBMA and IMAR^2 have been introduced. These strategies are based on a weighted lottery algorithm, which uses information given by hardware counters with low overhead to build the 3DyRM for guiding the scheduling and migration process. The final result is implemented as a runtime application, executed in background in user space, capable of handling dynamic workloads with several processes, without requiring any modification of the target code to be executed, monitored and migrated.

Our algorithms have been experimentally tested by simulating several concurrent users launching different NAS benchmarks, which are widely known, on two different platforms. Results show that even simple algorithms can increase performance significantly. LBMA and IMAR^2 strategies improved the execution time up to 38%, compared to the standard operating system scheduler in the performed benchmarks. Different variations of the IMAR^2 algorithm have also been considered, where only some parameters of the 3DyRM were used. These variations, while improving the operating system behaviour, did not reach the results obtained with the complete 3DyRM, proving the relevance of this model.

We plan to follow up this work by implementing several features, such as different types of scalarisations of 3DyRM information, or a procedure to dynamically change the value of the different tickets granted according to their relevance. Also, new multiobjective optimisation algorithms might be considered. In addition, page and memory migration may be needed to further improve results, since thread migration may not be enough to reach the optimal NUMA placement.

ACKNOWLEDGEMENTS

This work has received financial support from the Ministerio de Ciencia e Innovación within the project PID2019-104834GB-I00. It was also funded by the Consellería de Cultura, Educación e Ordenación Universitaria of Xunta de Galicia (accr. 2019-2022, ED431G2019/04 and reference competitive group 2019-2021, ED431C 2018/19).

References

1. Ju M, Jung H, Che H. A performance analysis methodology for multicore, multithreaded processors. *IEEE Tran. on Computers* 2014; 63(2): 276–289.

2. Chasparis GC, Rossbory M. Efficient Dynamic Pinning of Parallelized Applications by Distributed Reinforcement Learning. *International Journal of Parallel Programming* 2017. doi: 10.1007/s10766-017-0541-y

3. Adhianto L, Banerjee S, Fagan M, et al. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 2010; 22(6): 685–701.

4. Cheung A, Madden S. Performance profiling with EndoScope, an acquisitional software monitoring framework. *Proc. of the VLDB Endowment* 2008; 1(1): 42–53.

5. Geimer M, Wolf F, Wylie BJN, Ábrahám E, Becker D, Mohr B. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 2010; 22(6): 702–719.

6. Schulz M, de Supinski BR. PNMPI tools: a whole lot greater than the sum of their parts. In: SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing. ACM/IEEE. ; 2007: 1-10

7. Williams S, Waterman A, Patterson D. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 2009; 52(4): 65–76.

8. Lorenzo OG, Pena TF, Cabaleiro JC, Pichel JC, Rivera FF. 3DyRM: A dynamic roofline model including memory latency information. *The Journal of Supercomputing* 2014; 70(2): 696–708.

9. Gureya D, Neto J, Karimi R, et al. Bandwidth-aware page placement in NUMA. *arXiv preprint arXiv:2003.03304* 2020.

10. Li T, Ren Y, Yu D, Jin S. Analysis of NUMA effects in modern multicore systems for the design of high-performance data transfer applications. *Future Generation Computer Systems* 2017; 74: 41 - 50. doi: https://doi.org/10.1016/j.future.2017.04.001

11. Agung M, Amrizal MA, Egawa R, Takizawa H. DeLoc: A locality and memory-congestion-aware task mapping method for modern NUMA systems. *IEEE Access* 2020; 8: 6937-6953.

12. Popov M, Jimborean A, Black-Schaffer D. Efficient Thread/Page/Parallelism Autotuning for NUMA Systems. In: ICS '19. ICS. Association for Computing Machinery; 2019; New York, NY, USA: 342–353

13. Drebes A, Pop A, Heydemann K, Cohen A, Drach N. Scalable task parallelism for NUMA: A uniform abstraction for coordinated scheduling and memory management. In: PACT '16. ACM. Association for Computing Machinery; 2016; New York, NY, USA: 125–137

14. Dashti M, Fedorova A, Funston J, et al. Traffic management: A holistic approach to memory placement on NUMA systems. *SIGPLAN Not.* 2013; 48(4): 381–394. doi: 10.1145/2499368.2451157

15. Chiang ML, Yang CJ, Tu SW. Kernel mechanisms with dynamic task-aware scheduling to reduce resource contention in NUMA multi-core systems. *Journal of Systems and Software* 2016; 121: 72 - 87. doi: https://doi.org/10.1016/j.jss.2016.08.038

16. Chiang ML, Tu SW, Su WL, Lin CW. Enhancing Inter-Node Process Migration for Load Balancing on Linux-Based NUMA Multicore Systems. In: . 2. IEEE. ; 2018: 394–399.

17. Chiang ML, Su WL, Tu SW, Lin ZW. Memory-aware kernel mechanism and policies for improving internode load balancing on NUMA systems. *Software: Practice and Experience* 2019; 49(10): 1485–1508.

18. Lepers B, Quema V, Fedorova A. Thread and memory placement on NUMA systems: Asymmetry matters. In: USENIX Association. USENIX Association; 2015; Santa Clara, CA: 277–289.

19. Intel Corp . Intel 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/articles/intel-sdm; 2017. [Online; Dec. 2019].

20. Eranian S. *Perfmon2: a standard performance monitoring interface for Linux*. HP Labs; HP Labs: 2008. http://perfmon2.sf.net/perfmon2-20080124.pdf.

21. Lorenzo OG, Pena TF, Cabaleiro JC, Pichel JC, Rivera FF. Multiobjective optimization technique based on monitoring information to increase the performance of thread migration on multicores. In: Cluster Computing (CLUSTER), 2014 IEEE Int. Conf. on. IEEE. ; 2014: 416–423.

22. Akiyama S, Hirofuchi T. Quantitative evaluation of Intel PEBS overhead for online system-noise analysis. In: Proc. of the 7th Int. Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017. ACM. ACM; 2017; New York, NY, USA: 3:1–3:8

23. Intel Developer Zone . Fluctuating FLOP count on Sandy Bridge. http://software.intel.com/en-us/forums/topic/375320; 2013. [Online; Nov. 2019].

24. Terpstra D, Jagode H, You H, Dongarra J. Collecting performance data with PAPI-C. In: Tools for High Performance Computing 2009. Springer. Springer Berlin Heidelberg; 2010; Berlin, Heidelberg: 157–173.

25. Lorenzo OG, Laso R, Pena TF, Cabaleiro JC, Rivera FF, Lorenzo JA. A new hardware counters based thread migration strategy for NUMA systems. In: 13th International Conference on Parallel Processing And Applied Mathematics. PPAM. ; 2019.

26. Huband S, Hingston P, Barone L, While L. A review of multiobjective test problems and a scalable test problem toolkit. *IEEE Transactions on Evolutionary Computation* 2006; 10(5): 477–506.

27. Braun M, Heling L, Shukla P, Schmeck H. Multimodal Scalarized Preferences in Multi-Objective Optimization. In: GECCO '17. GECCO. Association for Computing Machinery; 2017; New York, NY, USA: 545–552

28. Hwang CL, Syed A, Masud M. *Multiple objective decision making, methods and applications: a state-of-the-art survey*. Springer-Verlag . 2012.

29. Keeney R. Decision analysis: and overview. *Operations Research* 1982; 30(5): 803–838.

30. Kleen A. A NUMA API for Linux. *Novel Inc* 2005.

31. McCalpin JD. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 1995; 1995: 19–25.

32. Jin H, Frumkin M, Yan J. The OpenMP implementation of NAS parallel benchmarks and its performance. tech. rep., Technical Report NAS-99-011, NASA Ames Research Center; : 1999.

33. Lameter C. NUMA (Non-Uniform Memory Access): An overview. *ACM Queue* 2013; 11(7): 40.

34. Rane A, Stanzione D. Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems. In: Proc. of 10th LCI Int'l Conference on High-Performance Clustered Computing. ; 2009.

How to cite this article: R. Laso, O. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. A. Lorenzo (2020), LBMA and IMAR^2: Weighted lottery based migration strategies for NUMA multiprocessing servers, *Journal*, *Volume*.