CISTER
Research Centre in
Real-Time & Embedded
Computing Systems

Conference Paper

Server Based Task Allocation to Reduce Inter-Task Memory Interference in Multicore Systems

Syed Aftab Rashid

CISTER-TR-191002

2019/12/16

Server Based Task Allocation to Reduce Inter-Task Memory Interference in Multicore Systems

Syed Aftab Rashid

CISTER Research Centre

Polytechnic Institute of Porto (ISEP P.Porto)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail: syara@isep.ipp.pt

https://www.cister-labs.pt

Abstract

In multicore systems tasks running on one coremay experience inter-task interference from tasks running onother cores. This inter-task interference is due to contention inusing shared resources such as caches, system bus and the mainmemory. In this work, we focus on one of the major sources ofcross-core interference in multicore systems, i.e., main memory.The idea is to allocate tasks to cores in a way that the totalmemory demand of all tasks executing at a time instant t is lessthan the minimum available memory bandwidth, i.e., DRAMmin.The problem is formulated as a server-to-core mapping problemwhere each server constitute a set of tasks corresponding toan application. As mapping problems in multicore systems areNP-hard, we use different heuristic and meta-heuristic basedapproaches to find a feasible solution. Results show that ourapproach can perform well in multicore systems with _ 8processing cores with the memory demand of each server upperbounded by DRAMmin=2.

Server Based Task Allocation to Reduce Inter-Task Memory Interference in Multicore Systems

Syed Aftab Rashid
CISTER, ISEP, Polytechnic Institute of Porto, Portugal

***Abstract*—In multicore systems tasks running on one core may experience inter-task interference from tasks running on other cores. This inter-task interference is due to contention in using shared resources such as caches, system bus and the main memory. In this work, we focus on one of the major sources of cross-core interference in multicore systems, i.e., main memory. The idea is to allocate tasks to cores in a way that the total memory demand of all tasks executing at a time instant t is less than the minimum available memory bandwidth, i.e., $DRAM_{min}$. The problem is formulated as a server-to-core mapping problem where each server constitute a set of tasks corresponding to an application. As mapping problems in multicore systems are NP-hard, we use different heuristic and meta-heuristic based approaches to find a feasible solution. Results show that our approach can perform well in multicore systems with ≤ 8 processing cores with the memory demand of each server upper bounded by $DRAM_{min}/2$.**

I. Introduction

Tasks executing on a multicore system may contend with each other to access shared resources such as last level cache, system bus and the main memory. The interference generated in accessing these shared resources can affect the worst-case execution/response time (WCET/WCRT) of tasks. For example, two tasks τ_i and τ_j concurrently running on two different cores of a multicore system may simultaneously need to load data/instructions from the main memory. However, main memory requests generated by τ_i may be delayed because the memory controller was busy serving the memory requests generated by τ_j. This delay in serving the main memory requests of τ_i may result in increasing its WCET/WCRT.

Several techniques have been proposed in literature both at hardware and software level to eliminate/minimize main memory contention delay suffered by tasks executing on a multicore system. Hardware based approaches [1]–[3] usually focus on designing customized memory controllers that can be used to reduce inter-task interference at the main memory. Software based techniques such as DRAM bank partitioning [4], [5] and memory bandwidth reservation [6], [7] are proposed to isolate memory requests generated by tasks running on different cores of a multicore processor in order to reduce main memory interference. Several other works in literature [4], [8]–[11] target different hardware components in order to estimate/minimize inter-task interference. It has been identified in State-of-the-Art (SoA) [12] that the inter-task interference suffered by tasks in a multicore system mainly depends on two factors, i.e., workload and hardware configuration. Workload defines the number of tasks that may execute simultaneously with the task under consideration and hardware configuration represents how different hardware resources are configured, i.e., shared or partitioned cache, bus arbitration protocol, memory bandwidth reservation etc. In this work we will focus on the workload property of execution environment to bound main memory interference suffered by tasks. It is proved in [6] that if the total number of main memory requests generated by all concurrently running tasks at any time instance t are less than or equal to the minimum available DRAM bandwidth the average main memory access latency of tasks executing on the multicore platform can be bounded. We build on the work in [6] to allocate tasks to cores in a manner that the main memory demand of concurrently executing tasks at any time instant t is kept as close to the minimum available DRAM bandwidth as possible. We model main memory contention problem as a task allocation problem. The modeling is achieved by using the concept of notional processors/servers [13], [14], where each task is first allocated to a notional processor/server using a suitable bin packing heuristics [15]. We realize that an optimal allocation of servers/tasks to cores is proven to be NP-hard in strong sense [15]. Moreover, for the problem at hand, the situation is even more complex as in this case we are not only interested in assigning servers/tasks to different cores but also in minimizing the total number of main memory accesses due to simultaneous execution of serves on different cores. This indeed complicates the problem, since we not only have to figure out on which core a server may execute but also at what time it may execute in order to reduce memory interference between servers as well as ensuring schedulability.

The main contributions of this work are as follows: (1) We model the main memory contention problem as a task allocation problem using the concept of notional servers. (2) We propose a heuristic, i.e., first-fit decreasing memory demand (FFDM), that uses memory demand as the primary criterion while mapping servers to cores. (3) We present two neighborhood search algorithms to improve the initial solution produced using FFDM. (4) Lastly, we present a meta-heuristic based on Simulated Annealing (SA) to diversify the search space and achieve global optima. Experimental results show that the proposed heuristic and meta-heuristic based approaches can perform well in multicore systems with ≤ 8 processing cores with the memory demand of each individual server upper bounded by $DRAM_{min}/2$.

II. Related Work

Optimal allocation of tasks to cores is proven to be NP-hard in strong sense [16]. Different non-optimal heuristics are usually used for task allocation. Dhall et al. [15] proposed two heuristics based on task periods i.e. rate monotonic next-fit and rate monotonic first-fit. In these heuristics tasks were sorted in non-increasing order of their periods and assigned

to cores. Davari et al. [17] provided a variation of these heuristics by ordering tasks w.r.t their utilization in descending order. Lakshmanan et al. [18] developed a set of partitioning bin-packing algorithms to deploy groups of communicating tasks on different processors to reduce the bandwidth required for communication between tasks. However, all the works mentioned above do not consider the inter-task interference due to task allocations and are explicitly aimed to provide higher utilization bounds by effectively partitioning tasks among different cores of a multicore processor. On the other hand, most of the work done with regard to reducing inter-task interference is done is context of hardware configurations i.e. where hardware components such as cache, bus and memory are configured in way that results in minimizing the level of inter-task interference. Akesson et al. [1], Paolieri et al. [2] and Reineke et al. [3] proposed memory controller that can be configured to serialize memory requests of tasks running on different cores to reduce interference inside the memory controller. Wu et al. [19] presented worst-case Dynamic Random-Access Memory (DRAM) accesses latency in multicore environment with separate DRAM banks being assigned to different cores. Similarly, Kim et al. [4] proposed an efficient bank partitioning scheme to estimate worst-case memory access delay for tasks running on different cores using private/shared DRAM banks. Yun et al. [6], [7] and Pellizzoni et al. [20] presented a memory throttling mechanisms to bound the number of memory requests generated by each core. Their work focused on memory bandwidth isolation between cores. All these approaches are difficult to implement without having the exact knowledge of the underline hardware. Most relevant to our approach are the work done by Aydin and Yang [21], Paolieri et al. [12], Lakshmanan et al. [18] and Muralidhara et al. [22]. In [21] authors proposed an energy-ware partitioning of tasks on a multicore platform to achieve taskset feasibility as well as energy optimization. The work is different from the work done in this paper as they only emphasize on reducing power consumption of the processor by balancing the task load on all the cores in a partitioned system. Moreover, our work focuses on memory interference optimization which may not result from a balanced division of workload across all cores. Similarly, the work presented in [12] proposes an interference aware task allocation by considering a set of different WCETs of tasks under different execution environment. Their work is focused on reducing the number of resources i.e. the utilization and cache size under different execution behaviors in a non-preemptive system. Whereas in this work, we consider a preemptive system and our focus is on reducing memory interference along with ensuring feasibility of tasks running on different cores. The work in [22] maps applications to different memory channels (banks) to reduce inter-application interference. Their approach is similar to work presented in [4], [19] which also targets a specific hardware configuration to reduce memory interference. Similarly, the work in [18] propose allocation of tasks with shared resources (logical) i.e. tasks are not independent and data is shared among them, which is not the case with our approach as we assume that tasks are independent and do not share any data between them.

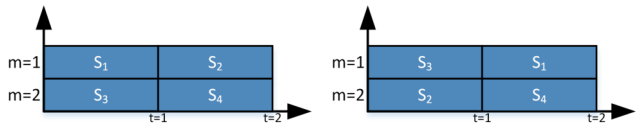
III. System Model

We consider a multicore platform with m identical cores, i.e., $\pi_1, \pi_2, ...\pi_m$. The set of cores is defined as Π. A taskset Γ consists of n tasks, i.e., $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$ where each task τ_i is defined by a quadruple (C_i, MD_i, T_i, D_i). C_i is the worst-case execution time (WCET) of τ_i, MD_i is the worst-case main memory demand of τ_i, i.e., the number of times τ_i may accesses main memory during its execution, T_i is the minimum inter-arrival between any two instances (or jobs) of τ_i and D_i is the relative deadline of τ_i. We assume task deadlines are implicit, i.e., $D_i = T_i$. The processor utilization of a task τ_i is denoted by U_i where $U_i = \frac{C_i}{T_i}$. For scheduling purposes, we use a server/notional processor based approach proposed in [13], [14]. We assume that tasks corresponding to one application are assigned to a single server S_i, i.e., a set of tasks. The set of all servers is denoted by S. The utilization of a server S_i denoted by U_{si} and is given by the sum of the utilizations of all tasks assigned to that server, i.e., $U_{si} = \sum_{\forall i \in S_i} U_i$. Where $U_{si} \leq 1$ and $U_S \leq m$. The total main memory demand of a server S_i is given by the sum of the memory demand of all tasks assigned to that server, i.e., $MD_{si} = \sum_{\forall i \in S_i} MD_i$. We consider partitioned scheduling approach where servers are statically partitioned among cores and once a server is assigned to a core it is not allowed to migrate. All servers execute inside a *periodic reserve* P. A periodic reserve is a fixed length time window available every P time units where P is given such that $P = \min(T_1, T_2, ...T_n)$. We assume that Earliest Deadline First (EDF) scheduling algorithm is used to schedule tasks inside a server. It is proved in [14] that implicit-deadline tasks of cumulative utilization U_{si} will always be schedulable under EDF if the length of the periodic reserve is equal to $U_{si} \times P$. This is also the reason of using a server based mapping since all tasks assigned to a server S_i will always be schedulable (under EDF) as long as the utilization based schedulability condition holds ($U_{si} <= 1$).

IV. Motivational Example and Problem Formalization

Main memory requests generated by a server S_i running on one core of a multicore processor can be affected by the memory requests generated by other servers running on other cores. This may result in increasing the memory access latency of S_i which effectively results in increasing the WCET/WCRT of tasks executing within S_i. However, if the total number of main memory requests generated by all the servers in a time window of length t are less than or equal to the minimum available DRAM bandwidth the memory access latency a server may suffer can be bounded and is no larger than when it is running in isolation on a dedicated albeit slower memory system.

What we aim in this work is to propose a suitable allocation heuristic that allocate servers to different cores such that the total main memory demand of all concurrently running servers at any time instance t is minimized and is less than or equal to the minimum available DRAM bandwidth denoted by $DRAM_{min}$. The allocation problem we are considering is more complex in comparison to a traditional bin-packing



(a) Server allocation using FFDU (b) Server allocation considering memory demand

Fig. 1: Example server-to-core allocation to highlight the importance of considering memory demand of servers

problem. For this problem, we cannot simply partition servers among cores based on their utilization but we also have to take into account the number of main memory requests generated by each server and at what time each server should be executed, e.g., a general intuition is not to run memory intensive servers on two or more cores at the same time. To illustrate, consider the scenario depicted in Figure 1. We have four servers S_1, S_2, S_3 and S_4 with utilization and main memory demand of each server given by $U_{si} = \{0.5, 0.5, 0.5, 0.5\}$ and $MD_{si} = \{10, 4, 15, 8\}$ respectively. The total number of processor cores to allocate these servers are two, i.e., $m = 2$. We set the periodic reserve $P = 2$, i.e., each server needs to execute at least for 1 sec in order to be schedulable. We also assume that the minimum available DRAM bandwidth $DRAM_{min}$ of the memory controller is 20. Figure 1a shows a server-to-core allocation based on popular multicore task allocation heuristic, i.e., first-fit decreasing utilization (FFDU), where servers are ordered in a descending order based on their utilizations and are assigned to the first available processor core. We can see in Figure 1a that using FFDU server S_1 and S_3 (also S_2 and S_4) are allocated to different cores however they will execute in parallel. Since $P = 2$ seconds, so for $t = 1$ the total memory demand will be $MD_{s1} + MD_{s3} = 25$ whereas for $t = 2$ we have a total memory demand of 12, i.e., $MD_{s2} + MD_{s4} = 12$. As $DRAM_{min} = 20$, we can see that for $t = 1$, the DRAM will be overloaded which may result in increasing the execution time of server S_1 and S_3. Whereas, for $t = 2$ DRAM is effectively underutilized. Let us now consider a different allocation shown in Figure 1b. we can see in Figure 1b that for $t = 1$ the total memory demand of the system, i.e., $MD_{s2} + MD_{s3}$, is equal to 19 which is less than the minimum available DRAM bandwidth of 20. Similarly the memory demand in the next time slot (i.e., $t = 2$) is also less than the minimum supported memory demand of the system, i.e., $MD_{s1} + MD_{s4} = 18 \leq DRAM_{min}$. Therefore, for the server-to-core allocation in Figure 1b we can say that the memory access latency of each server can be easily bounded which is not true for the allocation shown in Figure 1a.

A. Problem Formalization

Our objective is to ensure that the memory demand of all servers executing during the periodic reserve is less than or equal to the minimum DRAM bandwidth $DRAM_{min}$. Let $MD_{tot}(t)$ denoted the total memory demand of all the servers executing concurrently at a time instant t, i.e.,

$$MD_{tot}(t) = \sum_{\forall S_i \text{executing at t}} MD_{si} \tag{1}$$

Effectively, our objective function OBJ can be defined over the periodic reserve P such that

$$OBJ = \min \sum_{t=1}^{P} (DRAM_{min} - MD_{tot}(t)) \tag{2}$$

We use a two-dimensional array of size $m \times P$ to represent server-to-core allocation, where m is the total number of processing cores and P is the value of the periodic reserve.

$$\begin{bmatrix} 1 & 2 & 3 & ... & P \\ . & 2 & 3 & ... & P \\ m & 2 & 3 & ... & P \end{bmatrix}$$

Using this above representation, the value of the objective function is determined using the function $mem\_demand()$ that calculate the memory demand of all the servers executing during an instant t inside the periodic reserve P and compare it with the minimum available DRAM bandwidth. During implementation we realized that the overall value of the objective function does not change significantly by applying different moves (that will be discussed in next section). Therefore, to better quantify the quality of server-to-core allocation we define a function $U(t)$

$$U(t) = \begin{cases} 1, & \text{if } DRAM_{min} \geq MD_{tot}(t) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$U(t)$ is represented by matrix of order $1 \times P$, i.e.,

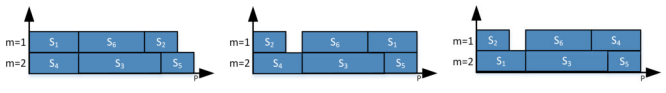
$$\begin{bmatrix} 1 & 0 & ... & P \end{bmatrix}$$

The solution with more numbers of 1's in matrix $U(t)$ is considered to be a better solution. If for two server-to-core allocations the matrix $U(t)$ is identical then we consider the solution with lower value of the objective function to be a better solution.

V. Initial Heuristic and Neighborhood Search

For an initial solution, we develop a heuristics similar to first-fit decreasing utilization (FFDU) and name it as first-fit decreasing memory demand (FFDM), i.e., servers are sorted by non-increasing memory demands and allocated to the cores using first-fit heuristic. Assuming we have n servers each defined by a tuple, i.e., server ID, utilization U_{sn} and MD_{sn}. FFDM will start by first ordering the servers such that $MD_{s1} \geq MD_{s2} \geq ... \geq MD_{sn}$ and then applying the first-fit bin packing. Initially all bins/processors are empty and we start with current processor m_i and server S_i. We consider all processors $1, 2, ..., m$ and place server S_i in the first processor that has sufficient available utilization. If no such processor is available we increment m and repeat until server S_n is allocated. Once the allocation is achieved function $mem\_demand()$ is used to calculate memory demand of all the servers that execute concurrently on different cores for every time instant t inside the periodic reserve P, where P is selected considering the server with minimum utilization, i.e., $P = \min(U_{s1}, U_{s2}, ...U_{sn})$. The resulting values from the $mem\_demand()$ function and the value of the minimum available DRAM bandwidth $DRAM_{min}$ are then used to generate matrix $U(t)$.

Example. Assume we have six servers $S_1, S_2, ..., S_6$



(a) FFDM (b) swap_servers(1,2) (c) swap_servers(1,4)

Fig. 2: Different server-to-core allocations

with each defined as $S_i = \{Id, MD_{si}, U_{si}\}$ to be allocated on two cores, i.e., $m = 2$. The set of servers S is given by $S = \{\{1, 50, 30\}\{2, 40, 20\}\{3, 20, 50\}\{4, 30, 30\}\{5, 20, 20\}\{6, 45, 40\}\}$, where utilization is the $\%$ of processor time required by the servers to execute. In first step, servers are sorted in non-increasing order of their memory demands. The sorted set of servers is given by $\hat{S}$, i.e., $\hat{S} = \{\{1, 50, 30\}\{6, 45, 40\}\{2, 40, 20\}\{4, 30, 30\}\{3, 20, 50\}\{5, 20, 20\}\}$. Servers in $\hat{S}$ are then assigned to cores using the first-fit decreasing memory (FFDM) heuristic as explained above. The resulting server-to-core allocation is shown in Figure 2a. To ensure schedulability of tasks executing within a server, each sever should execute in proportion to its utilization U_{si} inside the periodic reserve P, i.e., $U_{si}\%$ of P. As the minimum server utilization is 20 hence the value of the periodic reserve P will also be set to 20. Consequently, the execution time of each servers will be set using the expression $U_{si}\%20$, e.g., server S_1 must execute for 6 time units (i.e., $30\%20 = 6$) inside the periodic reserve $P = 20$ to ensure schedulability of all tasks assigned to server S_1. The initial solution resulting from FFDM heuristic for $m = 2$ and $P = 20$ is given by

$$\begin{bmatrix} s_1 & s_1 & s_1 & s_1 & s_1 & s_1 & s_6 & s_6 & s_6 & s_6 & s_6 & s_6 & s_6 & s_6 & s_2 & s_2 & s_2 & s_2 & x & x \\ s_4 & s_4 & s_4 & s_4 & s_4 & s_4 & s_3 & s_3 & s_3 & s_3 & s_3 & s_3 & s_3 & s_3 & s_3 & s_3 & s_5 & s_5 & s_5 & s_5 \end{bmatrix}$$

Using the above representation, the value of the objective function is determined by summing up memory demand of all servers executing at every time instant t inside the periodic reserve P. Assuming the minimum available DRAM bandwidth $DRAM_{min}$ is 60, the final formulation of function $U(t)$ for the server-to-core allocation of Figure 2a is given below

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

A. Moves

We have implemented two different type of moves, those lead to efficient implementation of neighborhood search algorithms. These moves are detailed as follows.

- Swap_severs(): This move takes as input id's of two servers that are executing on the same processor cores and swap them. It essentially changes the time at which two servers will be executed, e.g., applying $swap\_servers(1, 2)$ to the server allocation shown in Figure 2a will result in changing the time at which server S_1 and S_2 will be executed (see Figure 2b).
- Swap_processors(): $Swap\_processors$ is similar to $Swap\_severs$, but it changes the processors allocated to two servers. The motive behind this move is to change the state of two processors simultaneously in order to reach a more feasible solution. To apply this move it is necessary that the servers involved have equal utilization or the destined processors have the capacity to execute the swapped server, e.g., applying $swap\_processors(1, 4)$ to the allocation shown in Figure 2b results in a new allocation shown in Figure 2c.

```
int i=0; j=0;
while i≤TotalNumberofServers do
    i++; j=1;
    while j≤TotalNumberofServers do
        if i and j are on same procssor then
            swap_servers(i,j);
            check mem_demand;
            accept new solution if better else discard;
            j++;
        end
        if i and j are on different procssors then
            swap_processors(i,j);
            check mem_demand;
            accept new solution if better else discard;
            j++;
        end
    end
end
```

Algorithm 1: Single Swap based NS Algorithm1

B. Neighborhood Search (NS)

To find the best neighborhood solutions we have designed two algorithms based on the moves defined in the previous section.

1) NS Algorithm1: NS algorithm1 is based on single swap moves. Pseudo code for NS algorithm1 is given by Algorithm 1. NS Algorithm1 starts by selecting a server, e.g., S_1 and swap it with every other server running on the same or different core in the system using the above defined moves. For example, if the selected servers are on the same core swap_servers (id1, id2) is performed otherwise executing processors are changed for the selected servers using swap_processors (id1,id2). After performing any move the value of the objective function is calculated and compared with the current value of the objective function. If the new value is better than the current value then the initial solution will be changed, otherwise the solution remain unchanged.

2) NS Algorithm2: NS algorithm2 is based on moves that are similar to 2-opt move usually used in optimization problems. In this algorithm rather than just moving two servers, we move four servers at the same time. This adds more diversity to local search and improve results in comparison to NS algorithm1. Due to space constraints we omit the formal representation of NS algorithm2 in the paper. NS algorithm2 also start by selecting the first server and swap it with every other server in the system. But after the first swap, we choose the immediate next servers from the first swap and apply the same moves described earlier. The value of the objective function is updated after performing these moves using the same methodology as in Algorithm 1.

VI. Meta-Heuristic Approach

Heuristics based on iterative improvement start with an initial solution and keep on rearranging the solution as long as the solution keep on improving. This process continues unless no further improvement is found. However, these approaches may often get stuck into a local optimum rather than global optima. Therefore, to further improve the solution we need to diversify and may be start from a randomly generated configuration each time. To add diversification to our approach we used simulated annealing (SA) based meta-heuristic approach. The concept of simulated annealing comes from physical process of 'annealing', i.e., a process used to obtain low energy sates of a solid using a heat bath where temperature is varied from very

high to low, very slowly in order to get the required state [23]. Simulating annealing starts with a randomized state and in a polling loop it moves to neighboring states always accepting the moves that improve the value of the objective function while only accepting bad moves according to a probability distribution dependent on the "temperature", i.e., based on the difference between the initial feasible solution and the randomly generated solutions by SA, i.e., SA may allow to accept bad solutions. This is different from other local search based techniques that only accept solutions that are better than the initial solution. The simulating annealing approach used in this paper is presented in Algorithm 2. SA algorithm starts by defining the required parameters that are used to simulate the annealing process. An initial solution is generated and the value of objective function is calculated for the resulting solution. If the resultant solution is better than the previous solution we move to the new solution. Otherwise the algorithm calculates delta, i.e., the difference between values of the objective functions for the previous solution and the new solution. SA does not immediately reject the new solution but may keep it as long as the exponential value of $\epsilon^{-delta/temp}$ is greater than a randomly generated number between 0 and 1. By doing so it adds diversity to the search space, exploring the area that is not explored by NS algorithm1 and NS algorithm2. The idea is to escape the local minima in order to reach an optimum global solution.

```
Define variable required for SA Algorithm;
while temperature≥epsilon do
    GenerateRandomSol();
    if new solution is better then
        Accept new solution;
    else
        delta = InitialSol − NewSol;
        if RandomProb(0, 1) < ε^(−delta/temperature) then
            Accept new solution;
        else
            Discard new solution;
        end
    end
    Gradually Reduce SA temperature;
end
```

Algorithm 2: Simulated Annealing Based Meta-heuristic

VII. Evaluation and Results

For evaluation purposes, we simulated the intel Core2Quad processor with 4 physical cores ($m = 4$). As shown in [6] this platform has a minimum DRAM available bandwidth of 1.2 GB/s when all cores are running, i.e., $DRAM_{min}$= 1.2 GB/s or 1200 MB/s. The maximum available DRAM bandwidth is 8490 MB/s. We generated random set of servers based on SPEC2006 benchmarks [6]. The solution with lower value of objective function and higher values of $U(t)$ is considered a better solution. Separate set of servers with High(H), Medium(M) and Low(L) memory intensity are generated and evaluated using the proposed heuristics. Table I shows the results produced in this case. We can see in Table I that when we have only high and medium memory intensity tasks the proposed heuristics may not perform well. This is due to the limitation on the minimum available bandwidth of DRAM memory i.e. $DRAM_{min}$= 1.2GB/s. For the same set of servers, the results of the proposed approaches improve as the minimum available DRAM bandwidth is increased. We can observe this improvement from the last two entries in Table I, i.e., when the minimum available DRAM bandwidth is increased to 4.8GB/s, our approaches tend to perform better even with high memory intensity servers. Also in case of low memory intensity servers, i.e. $MD_{si} = 100\ 500$ MB/s, we can see significant improvements over the initial incumbent solution (i.e., FFDM heuristic).

TABLE I: Performance of proposed approaches against randomly generated High (H), Medium (M) and Low (L) memory intensity servers

Servers	Cores	Memory Intensity (MB/s)	$DRAM_{min}$ (MB/s)	FFDM obj.val, $U(t)$	NS Algo-1 obj.val, $U(t)$	NS Algo-2 obj.val, $U(t)$	SA Algo obj.val, $U(t)$
10	4	1000-2000 (H)	1200	63092, 0	63092, 0	63092, 0	63092, 0
20	4	1000-2000 (H)	1200	77916, 0	77916, 0	77916, 0	77916, 0
10	4	500-1000 (M)	1200	26156, 0	26156, 0	26156, 0	26156, 0
20	4	500-1000 (M)	1200	23900, 0	23900, 0	21900, 0	21900, 0
10	4	100-500 (L)	1200	2308, 16	1700, 20	1700, 20	1700, 20
20	4	100-500 (L)	1200	2228, 8	2324, 12	3364, 16	3364, 16
10	4	1000-2000 (H)	4800	2748, 12	4556, 16	4140, 16	548, 20
20	4	1000-2000 (H)	4800	7076, 0	7612, 4	8868, 8	7868, 8

TABLE II: Results derived using servers with memory intensity varying from 100 to 2000 MB/s

Servers	Cores	Memory Intensity (MB/s)	$DRAM_{min}$ (MB/s)	FFDM ($U(t)$)	NS Algo-1 ($U(t)$)	NS Algo-2 ($U(t)$)	SA Algo ($U(t)$)	Optimal value ($U(t)$)
10	4	100-2000	1200	2	4	5	12	20
20	4	100-2000	1200	0	4	8	12	20
30	4	100-2000	1200	0	4	4	8	20
40	4	100-2000	1200	0	1	3	4	10
50	4	100-2000	1200	0	1	2	3	10

TABLE III: Results derived by keeping memory demand of servers $\leq DRAM_{min}/2$

Servers	Cores	Memory Intensity (MB/s)	FFDM ($U(t)$)	NS Algo-1 ($U(t)$)	NS Algo-2 ($U(t)$)	SA Algo ($U(t)$)	Optimal value ($U(t)$)
10	2	$\leq DRAM_{min}/2$	4	8	12	20	20
20	4	$\leq DRAM_{min}/2$	2	8	12	16	20
30	6	$\leq DRAM_{min}/2$	0	4	8	12	20
40	8	$\leq DRAM_{min}/2$	0	2	4	10	20
50	10	$\leq DRAM_{min}/2$	0	1	2	3	20

Table II shows the experimental results by generating mix set of servers with different memory demands, i.e., high, medium and low, with memory demand of each server ranging between 100 MB/s to 2000 MB/s. The results show that the proposed heuristics work very well when we have servers with different memory intensities as often the case in real applications. The initial incumbent solution (FFDM) turn out to be the worst in each case. NS algorithm1 and NS algorithm2 perform similarly with the later dominating the former in most of the cases. We used the simulating annealing approach to a maximum of ten thousand iterations. However, we can see that simulated annealing performs better than the other approaches in all the cases. Note that we only show average results due to space constraint but the actual experiments were performed with 50-100 set of servers generated in every iteration. We can also see in Table I and I that as the number of servers increase the effectiveness of the proposed approaches decreases. This is due to the fact that as we increase the number of servers and processors the effectiveness of swap based moves that involve only two servers is reduced. In another experiment,

we generated set of servers with each having memory intensity bounded such that $\leq DRAM_{min}$, we observed that by setting a higher upper bound on memory intensity of servers as the number of servers increase the gap between the optimal and resultant solution also increased. We can also noticed that SA perform better than other two algorithms even when the number of servers are high. This effect is due to the diverse nature of the algorithm. We can also observed a similar situation by increasing the number of processor cores, i.e., m. For final phase of our evaluation we generated servers with memory intensity $\leq DRAM_{min}/2$. We observed that by having this limitation on the memory demand of each server, our proposed heuristic based approaches can work well with up to 8 processing cores with up to five servers running on each core. The results are shown in Table III, where we can see that the performance of the proposed approaches is less effected by the number of servers and processors but the gap between the optimal and resultant solution still increases with the increase in number of servers/processors.

VIII. Conclusion

In this work, we focused on main memory which is one of the major sources of cross-core interference in multicore systems. We modeled the memory contention problem as an allocation problem. A server based approach is used to ensure schedulability, where tasks are assigned to servers and are executed within those servers under EDF scheduling algorithm. As a first solution we proposed a simple heuristic FFDM for server-to-core mapping. Two neighborhood search algorithm are proposed to improve upon the initial solution generated by FFDM. Finally, a simulated annealing based meta-heuristic is used to diversify the search space and achieve global optima. Experiments show that the proposed approaches can generate solutions that are feasible as long as the number of server/cores are less than a certain threshold. We also concluded that to obtain better results memory demand of each server should be less than half of the minimum available DRAM bandwidth. In future, we will consider more diversified approach to solve this problem. We also plan to generate moves that can be applied to a number of servers at the same time. For the case of large number of server and processing core, the problem can be subdivided into separate problem where each can be optimized separately. Use of population based heuristics for this problem also present an interesting area for future research.

Acknowledgments

This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (UID/CEC/04234); by FCT and the ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/119150/2016.

References

[1] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable sdram memory controller," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*. IEEE, 2007, pp. 251–256.

[2] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, 2009.

[3] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2011 Proceedings of the 9th International Conference on*. IEEE, 2011, pp. 99–108.

[4] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 2014, pp. 145–154.

[5] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 367–376.

[6] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 55–64.

[7] ——, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 299–308.

[8] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*. IEEE, 2007, pp. 49–60.

[9] S. A. Rashid, G. Nelissen, D. Hardy, B. Akesson, I. Puaut, and E. Tovar, "Cache-persistence-aware response-time analysis for fixed-priority preemptive systems," in *ECRTS*, 2016, pp. 262–272.

[10] S. A. Rashid, G. Nelissen, S. Altmeyer, R. I. Davis, and E. Tovar, "Integrated analysis of cache related preemption delays and cache persistence reload overheads," in *RTSS*. IEEE, 2017, pp. 188–198.

[11] S. A. Rashid, G. Nelissen, and E. Tovar, "Trading between intra-and inter-task cache interference to improve schedulability." in *RTNS*, 2018, pp. 125–136.

[12] M. Paolieri, E. Quiñones, F. J. Cazorla, R. I. Davis, and M. Valero, "Iaˆ 3: An interference aware allocation algorithm for multicore hard real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011, pp. 280–290.

[13] K. Bletsas and B. Andersson, "Notional processors: an approach for multiprocessor scheduling," in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*. IEEE, 2009, pp. 3–12.

[14] ——, "Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound," *Real-Time Systems*, vol. 47, no. 4, pp. 319–355, 2011.

[15] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations research*, vol. 26, no. 1, pp. 127–140, 1978.

[16] R. G. Michael and S. J. David, "Computers and intractability: a guide to the theory of np-completeness," *WH Free. Co., San Fr*, pp. 90–91, 1979.

[17] S. Davari, "An on line algorithm for real-time tasks allocation," in *IEEE Real-Time Systems Symposium, 1986*, 1986, pp. 194–200.

[18] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*. IEEE, 2009, pp. 469–478.

[19] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of dram latency in multi-requestor systems," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013, pp. 372–383.

[20] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 2010, pp. 741–746.

[21] H. Aydin and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. IEEE, 2003, pp. 9–pp.

[22] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Microarchitecture (MICRO), 2011 44th Annual IEEE/ACM International Symposium on*. IEEE, 2011, pp. 374–385.

[23] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.