Evaluation of Memory Access Arbitration Algorithm on Tilera's TILEPro64 platform

Mayank Shekhar
Southern Illinois Univ Carbondale
Carbondale, IL U.S.A
mayank@siu.edu

Harini Ramaprasad
Univ of North Carolina at Charlotte
Charlotte, NC USA
hramapra@uncc.edu

Frank Mueller
North Carolina State Univ
Raleigh, NC U.S.A
mueller@cs.ncsu.edu

***Abstract*—As real-time embedded systems demand more and more computing power under reasonable energy budgets, multi-core platforms are a viable option. However, deploying real-time applications on multi-core platforms introduce several predictability challenges. One of these challenges is bounding the latency of memory accesses issued by real-time tasks. This challenge is exacerbated as the number of cores and, hence, the degree of resource sharing increases.**

Over the last several years, researchers have proposed techniques to overcome this challenge. In prior work, we proposed an arbitration policy for memory access requests over a Network-on-Chip. In this paper, we implement and evaluate variants of our arbitration policy on a real hardware platform, namely Tilera's TilePro64 platform.

I. Introduction

As demand for computation and concern for energy efficiency increase, multi- and many-core platforms are increasingly important even for real-time embedded systems. One of the many challenges that multi-core platforms pose to real-time operation is bounding predictability of memory accesses. As the number of cores on a single chip increases, contention for access to off-chip (main) memory increases. According to a study [1], a task suffers 300% increase in WCET with only 10% of its time spent on fetching memory on an eight core system.

Researchers have proposed several approaches to solve this problem. One option is to implement a memory arbitration scheme in software. Bellosa *et al.* [2] proposed a memory throttling scheme in the operating systems layer, for soft-real-time systems. While such a software-based approach has significant overhead that could reduce performance, it has the advantage of allowing the use of Commercial Off the Shelf (COTS) platforms. Yun *et al.* [3] propose a throttling mechanism to isolate the cores executing critical tasks from those running non-critical tasks. In this work, they assume a cache miss takes a constant amount of time, which may not be true in practice. Furthermore, the scalability of the approach is a concern. As the number of cores increases, the overhead would increase significantly.

Another option is to modify the hardware to allow efficient memory arbitration. This approach has the disadvantage that it requires specialized hardware. However, the overhead incurred is significantly less compared to software-based approaches. Akesson *et al.* [4] propose a predictable DRAM controller that may be used to provide guaranteed bounds on latency and bandwidth for off-chip memory access. In this work, the authors employ a hybrid approach between static and dynamic scheduling at the memory controller, but the contention on chip can also make external memory accesses unpredictable.

In recent work, we propose a dynamic off-chip memory arbitration scheme, EDF-on-NoC, that schedules off-chip memory accesses over a shared path on a Network-on-Chip (NoC) using an Earliest Deadline First (EDF) policy [5]. In that work, we conduct a *simulation* based study to demonstrate the effectiveness of our algorithm for two reasons. First, using a simulated environment allows us to perform several sensitivity studies using synthetic or real benchmarks. Second, although all the architectural features that are assumed in this algorithm exist on real hardware platforms, to the best of our knowledge, there is no *single* platform that has all the features we assume.

In order to analyze the practical factors involved in implementing an algorithm like the one proposed in our prior work [5], in this paper, we implement and analyze variants of the algorithm on Tilera's TilePro64 platform [6]. TilePro64 is a homogeneous 64-core architecture with a mesh-based Network-on-Chip (NoC) communication infrastructure. Each core has private (dedicated) L1 and L2 caches and there are no shared caches on the platform. Although TilePro64 does not have all the features that we assume, the homogeneity of cores, the existence of only private caches and the NoC topology fit our assumptions, thus motivating our choice.

II. EDF-on-NoC Algorithm

In prior work, we propose two memory access arbitration schemes, namely Weighted TDMA [7] and EDF-on-NoC [5], with the aim of improving predictability of off-chip memory accesses. Simulation results show that EDF-on-NoC outperforms WTDMA. In the current paper, we implement variants of our EDF-on-NoC policy on the TilePro64 platform.

The EDF-on-NoC policy assumes a homogeneous multi-core architecture, where each core has private, set associative, lockable caches, and a two dimensional (2D) mesh-based NoC interconnect with dedicated, bidirectional channels for cache-to-cache transfers between cores that do not interfere with channels for regular main memory accesses.

The fundamental goals of EDF-on-NoC are 1) to control the rates of memory requests issued by different cores

This work was supported in part by NSF grants CNS-0905212, CNS-0720496 and CNS-0905181.

to different pre-calculated values; and 2) to schedule the memory transfers over a predetermined set of NoC links using a dynamic priority scheduling policy. In this policy, memory requests from different cores experience different access latencies, depending on the cores distance from the memory controller.

Off-chip memory access requests from a core are modelled as a real time task with a pre-defined period and worst-case execution time (WCET). The *period* of a core refers to the minimum time interval between two consecutive off-chip memory access requests that the core is allowed to issue and its *WCET* refers to the time required to complete a single off-chip memory access, assuming no interference from any other core. Since a core can issue off-chip memory requests only at period intervals, the effective off-chip memory access latency experienced by the core is equal to its period.

We illustrate the working of EDF-on-NoC using an example. Figure 1 shows a group of four cores that share a memory controller port. It is assumed that all memory requests from cores in this group are directed in a straight line along the group to the memory controller. So, off-chip memory requests from core A go through cores B, C and D to the memory controller. The WCET of a core is the sum of the time taken for a memory request/response to travel on-chip from/to the core and the time taken for a request to be serviced after it reaches the memory controller. We refer to the former as *on-chip memory latency* and the latter as *off-chip memory latency*. The off-chip memory latency is assumed to be the same for memory requests issued by any core (an upper bound of 60 cylces in the case of TilePro64), but the on-chip memory latency depends on the location of the core on the chip. In our example, a memory read

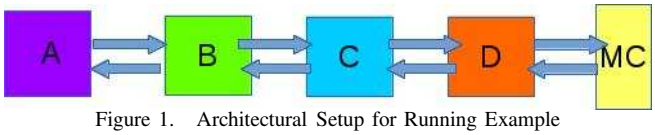


Figure 1. Architectural Setup for Running Example

request from core A travels as a single packet, leading to an on-chip latency of 4 cylces. A memory response, which is of memory line or cache line size, consists of four packets that travel in a pipelined fashion. This implies that the first packet has an on-chip latency of four cycles to get to core A and the rest of the packets reach in three successive cycles, bringing the total on-chip latency for the memory line to 7 cycles. The case of memory writes is analogous. Hence, the WCET for core A is calculated as 4 cycles + 60 cycles (off-chip latency) + 7 cycles = 71 cycles. Similarly, we can calculate the WCETs of cores B, C and D to be 69, 67 and 65, respectively. The period of a core is calculated based on the utilization and schedulability on the core. So, it depends on the scheduling policy used on the core.

III. Tilera's TilePro64 Platform

TilePro64 implements a two-dimensional array of processing elements or tiles. Each tile has a full featured processor that can independently run an operating system such as Linux, a dedicated cache engine and a switch engine. Tiles are connected to each other and to the external memory and I/O interfaces via multiple two-dimensional mesh networks.

The cache engine on each tile has a 16KB L1 instruction cache, 8KB L1 data cache and 64KB unfied L2 (instruction and data) cache. The switch engine is responsible for directing on-chip traffic using the NoC interconnect. There are five types of dynamic networks on TilePro64 that are used to dynamically route traffic on-chip. In this paper, we only consider the ones that are involved in routing off-chip memory traffic from and to the memory controller. Specifically, we use the Memory Dynamic Network or MDN that transfers packets from the queue of a tile to external memory and the Transfer Dynamic Network or TDN that transfers packets from external memory to a tile's cache memory.

The software stack running on TilePro64 consists of two layers. The lower layer is the hypervisor, which is responsible for the low-level hardware interface tasks and their resource allocation. The upper layer contains the supervisor instance(s). Although the platform allows different instances of the supervisor on each tile, we use a single instance of a VM-Linux supervisor to control multiple tiles in our current work. While TilePro64 supports most of the features required by the EDF-on-NoC policy, there are two important aspects that it does not support, thus preventing a direct implementation of EDF-on-NoC as proposed. Specifically, it does not support cache locking and controlled external memory access.

Cache locking is a technique that may be used to improve the timing predictability of real-time tasks. The idea is that a task may explicitly load and lock pre-determined content into the cache. For the duration that the content is locked, cache behavior becomes completely predictable. TilePro64 does not support cache locking.

The EDF-on-NoC algorithm assumes that a line of cores access a common memory controller port, as shown in Figure 1, and that we have control over the actual issue of memory requests. On the TilePro64 platform (shown in Figure 2), although we can limit access of a line of cores to a given memory controller port, we do not have control over the MDN or TDN that are responsible for carrying off-chip memory traffic. Instead, in case of a cache miss, hardware directly triggers an off-chip memory request.

IV. Methodology

In the previous section, we described the architectural limitations of TilePro64 that prevent a direct implementation of the EDF-on-NoC [5] algorithm on it. In this section, we describe a work-around that may be used to demonstrate the effect of our algorithm on such a system.

Since we do not have control over the MDN and TDN networks that handle off-chip memory requests, we need to control the issue of requests from the core itself. In order

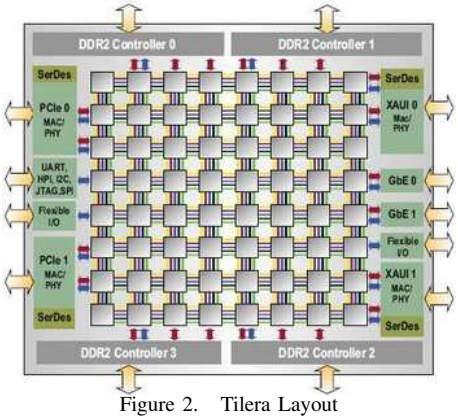


Figure 2. Tilera Layout

to achieve this, we need to have *a-priori* knowledge of what memory access requests will result in a cache miss and, hence, an off-chip access. Furthermore, since EDF-on-NoC arbitrates NoC usage among cores that share a single memory controller port, a single instance of an operating system must be deployed to control all cores sharing a single memory controller port.

On a system with lockable caches, it is straightforward to know which accesses will result in cache misses and which ones will not. However, since TilePro64 does not support cache locking, we resort to a slightly different approach. We *disable caching* for those memory regions that are not chosen to be locked in the cache so that all accesses to those regions are guaranteed to result in an off-chip memory access. This suffices for now since our primary focus is to demonstrate the shared NoC arbitration policy.

Our basic method for arbitrating off-chip memory access requests over the NoC consists of the following steps.

- Before issuing a memory access request for data that is known not to be in cache (i.e., one that will result in an off-chip memory access) a task sends a request to a central scheduler.
- After sending a request to the central scheduler, the task busy-waits until its off-chip memory access request is scheduled. This prevents multiple off-chip memory requests from one core in the central scheduler.
- The central scheduler schedules off-chip memory access requests according to whatever policy we choose to implement within it.

The above method can only control the release of an off-chip memory request from a core, but once it has been issued, cannot preempt a request at any point on the path to off-chip memory. However, EDF-on-NoC [5] was proposed as a *preemptive* scheduling policy over the shared NoC. Off-chip memory access requests were modelled as real-time tasks and these (virtual) real-time tasks were scheduled using EDF on the shared path to the memory controller. In order to implement EDF-on-NoC on TilePro64, we need to modify the algorithm to use *non-premptive* EDF [1].

There are two possible ways of implementing the central off-chip memory access request scheduler, namely in kernel space and in user space. We have implemented both on TilePro64.

A. Kernel-space implementation

The scheduler is implemented as a kernel module on the system. It executes in an infinite loop and maintains a queue of off-chip memory requests from the sharing cores. The scheduler can be started and stopped using system call.

Submission of a request for off-chip memory access to the scheduler is done via a system call that queues up the request in a common ready queue. The signalling mechanism in VM-Linux is used for the kernel to notify the corresponding user-space thread when it may issue its request.

1) Pros and cons: The advantage of using a kernel-space scheduler is that it provides a generic platform for the execution of all user-space threads. It is independent of the program executing in user space. The disadvantage of using a kernel-space scheduler is that there is an additional overhead of system calls added to the execution time of user-space threads. This additional overhead is so large that it overshadows any benefits of the implemented NoC arbitration algorithm.

B. User-space implementation

In the user space, the scheduler is implemented as a thread. A core is dedicated to this thread so that it does not interfere with the execution of tasks executing as user-space threads on other cores. The scheduler executes in an infinite loop. It is started by the main() function before task threads are spawned. It is also aborted by the main() function once all task threads finish their execution. A task thread places an off-chip memory access request in a common queue. Since this queue is shared, a mutex variable is used to protect it. The scheduler communicates with the scheduled task thread using a shared variable, also protected via a mutex variable.

1) Pros and cons: As explained before, an off-chip memory scheduler in user-space is implemented as a thread dedicated to a core. Since function calls rather than system calls are used to interact with the scheduler, the overhead of a user-space scheduler is significantly less than that of a kernel-space scheduler. The disadvantage of using such a scheduler is that we need to integrate it with the multi-threaded task at the user level. In other words, the main function that spawns other tasks as threads is also responsible for spawning the scheduler thread.

[1]The purpose of the current paper is to *practically* demonstrate and evaluate a dynamic-priority policy for off-chip memory access arbitration. A discussion on the schedulability analysis for a non-preemptive version of EDF is out of the scope of this paper

C. Practical implementation issues for user-space scheduler

In this paper, we present the results for the user-space scheduler implementation, but not the kernel-space one because of the prohibitive overhead of the system calls incurred by the latter implementation. Although the user-space implementation of the scheduler incurs much less overhead, there are some practical factors that undermine its performance, as discussed next.

For the TilePro64 platform, the maximum of the minimum relative deadline for off-chip memory access from a core is equal to 71 processor cycles (7+4 on-chip and 60 off-chip latency). This calculation is similar to the WCET calculation discussed earlier and is explained in more detail in our prior work [5]. We use *get_cpu_cycles()* in order to get the current processor cycle of the system. Unfortunately, the overhead of this function call is approximately 200 cycles. So, our scheduler can not operate at the granularity ideally required.

So, in order to be able to implement non-preemptive EDF-on-NoC scheduling, we increase the deadlines of the off-chip memory accesses by 200. This results in a decrease in utilization of the shared path to the off-chip memory and hence deteriorates the performance of the policy.

In order to demonstrate the effect that non-preemptive EDF-on-NoC should ideally have if the scheduler could operate at the desired granularity, we implement a Round Robin policy in the scheduler. The Round Robin policy is implemented using mutex variables. Requests for off-chip memory access from different cores are placed in a FIFO queue and arbitrated by the scheduler in a non-preemptive Round Robin fashion. Each task signals the completion of its off-chip memory access request by setting a mutex variable. Since there is no concept of deadline for off-chip memory access requests here, we do not use *get_cpu_cycles()* for deadline comparison. Hence, this implementation does not suffer from the granularity limitation. In our experimental setup, we use tasks with identical charecteristics, namely deadline, WCET, period and initial phase. In this case, non-preemptive EDF would also schedule tasks one by one in a queue, thus making it the same as a round robin policy.

V. Experimental Setup

We implement our algorithm on a set of four cores (e.g., cores A,B,C and D In Figure 1), all of them lined up to be on a common path to the memory controller. In order to achieve this on Tilepro64, we use a hypervisor configuration file where we power on the four cores shown in Figure 1. We also use a *non-striped* external memory structure. A *non-striped* memory structure implies that we can access any part of the external memory through any of the memory controllers. We then disable all memory controllers except the one that is shared by the four cores of interest. As mentioned before, VM-Linux is used as an operating system on the given set of cores.

Tasks are implemented as multiple threads of a program. Each thread is bound to a single core and only one thread resides on each core. All threads are identical in functionality and data sizes. This is done so that we can analyze and compare the execution times of each task allocated to a different core. In each thread, we first allocate an integer array of a given size. Next, we have two loops. In the first loop, we write random numbers into each element of the array. In the second loop, we read each element of array. We disable caching of the allocated integer array. Thus, every access to the integer array would lead to an off-chip memory access request. The execution time of a thread in our experiments is defined as the time taken by the two loops in each thread to finish execution. We use a built-in function called $get\_cpu\_cycles()$ in the VM-Linux to calculate the time interval between the start and end of a loop.

VI. Experimental Results

We conducted experiments with the tasks described in the previous section with varying data set sizes. We used an integer array size of 5000, 10000 and 15000 elements. For every data set size, we conducted 100 experiments. In each experiment, we measure the execution time of all threads that are allocated to separate cores.

In the experiments, we compare the execution times of the tasks when we do not use any specific bus arbitration policy with those when we use (non-preemptive) EDF-on-NoC and Round Robin as the underlying policy. Since the purpose of any bus arbitration policy is to make the execution times predictable, i.e., decrease the variation in execution times, we first compare the standard deviation of execution times in all the three cases.

A. Comparing Execution Times: Standard Deviation

Figure 3 depicts the result of the comparison of the standard deviation of execution times calculated in all the three cases (no policy, EDF-on-NoC and Round Robin). The X-axis represents the number of elements used in the integer array of the experiments and Y-axis represents the standard deviation of execution times.

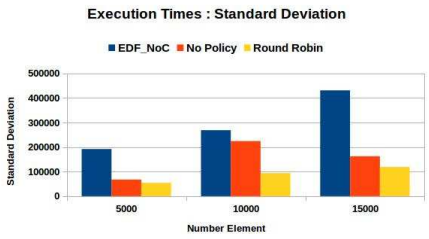


Figure 3. Comparing Standard Deviation of Execution Time

We observe that, EDF-on-NoC demonstrates the maximum standard deviation of execution time although it has been theoretically proved to be a better bus arbitration scheme (as shown in [5]). This is because we are not able to get the granularity of measuring processor cycles as required by EDF-on-NoC as explained in Section IV-C. In contrast, we observe that Round Robin, which theoretically exhibits

a similar behaviour to EDF-on-NoC, has minimum standard deviation in execution times and hence makes execution time more predictable.

Since we implement our arbitration policies (EDF-on-NoC and Round Robin) in software, they impose additional overhead on the execution times of the tasks. So, comparing the standard deviation of execution times of the tasks in all the three cases does not reflect a fair comparison of the performance of the policies themselves. In order to fairly compare the variation in execution times, we define a parameter called Variation Ratio.

The Variation Ratio in execution times for a given experiment is defined as the ratio of the difference between the maximum and minimum execution times to the minimum execution time for that experiment. This is mathematically represented by Equation 1.

$$Execution\_Variation = \frac{Execution_{max} - Execution_{min}}{Execution_{min}} \quad (1)$$

In Equation 1, $Execution_{max}$ and $Execution_{min}$ represents the execution times of the tasks *excluding* the implementation overhead in each case.

In order to estimate this overhead, let us denote the shortest execution time of the tasks when we do not use any policy as $exec_{base}$ and the shortest execution time of the tasks when we use a given policy (EDF-on-NoC or Round Robin) as $exec_{policy}$. Total overhead is the difference between $exec_{policy}$ and $exec_{base}$. Overhead per data element can then be expressed mathematically by Equation 2.

$$Overhead_{per\_element} = \frac{exec_{policy} - exec_{base}}{Data\_Size} \quad (2)$$

B. Comparing Variation Ratio of Execution Time

In all the graphs shown below, the X-axis represents the experiments and the Y-axis represents the execution times of the tasks allocated to different cores for each experiment. In each experiment we show the execution of tasks without any off-chip memory access policy, with EDF-on-NoC and with Round Robin as the underlying off-chip memory access policy with varying size of the integer array.

1) Experiment 1 — 5000 integers: Figure 4 shows the result for an integer array of 5000 elements and we observe that $exec_{base}$ is approximately equal to 800000, $exec_{edf\_noc}$ is approximately equal to 2900000 and $exec_{round\_robin}$ is 3600000. The additional overhead for 5000 data elements is equal to the difference of $exec_{policy}$ and $exec_{base}$. From Equation 2, the overhead incurred per element, represented as $Overhead_{per\_element}$, for implementing EDF-on-NoC is 400 and that for Round Robin is 560. [2]

Next, we calculate and compare the variation ratio in execution time of tasks in all the cases. From Figure 4, we get the variation ratio in execution time for the experiment with no policy as $\frac{1400000-800000}{800000}$, which is equal to 0.75, for EDF-on-NoC as $\frac{1500000-1000000}{1000000}$ which is equal to approximately 0.5 and for Round Robin as $\frac{1070000-850000}{850000}$, which is equal to 0.25. Thus we see that by using EDF-on-NoC on TilePro64, the variation ratio in execution time reduces from 0.75 to 0.5, which further reduces to 0.25 by using Round Robin.

2) Experiment 2 — 10000 integers: Figure 4 shows the result for an integer array of 10000 elements. In this experiment, the $Overhead_{per\_element}$ for EDF-on-NoC is calculated as 430 and that of Round Robin is 550. Next we calculate the variation ratio in execution time of the tasks using Equation 1. The variation ratio in execution time for the experiment using no policy for the off-chip memory access is calculated as 0.5 and that using EDF-on-NoC policy is calculated as approximately 0.375. The variation ratio further decreases to 0.235 in case of Round Robin. Thus in this experiment, we again observe a decrease in the variation ratio of execution times when we use EDF-on-NoC and Round Robin.

3) Experiment 3 — 15000 integers: Figure 4 shows the result for an integer array of 15000 elements. The $Overhead_{per\_element}$ is calculated as 370 for EDF-on-NoC and 553 for Round Robin from Figure 6. Thus, we observe in all the experiments that the overhead of implementation of both the policies remain very similar.

Next, we calculate the variation ratio in execution times for the three cases shown in Figure 6. Using Equation 1, we get the variation ratio in execution times as 0.54, when we do not use any underlying off-chip memory access policy which decreases to 0.44, when we use EDF-on-NoC as the off-chip memory access policy and further decreases to 0.20 when we use Round Robin for the same.

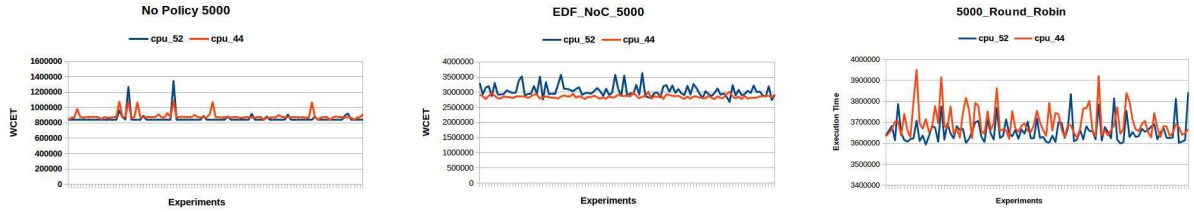
Thus, we observe that in all the experiments, we achieve a decrease in variation ratio in execution times by implementing a NoC arbitration scheme on TilePro64. We also established that there is a consistent overhead of implementing both EDF-on-NoC and Round Robin schemes in software.

VII. Conclusion

In this work, we implement variants of the EDF-on-NoC memory request arbitration scheme proposed in prior work on Tilera's TilePro64 platform. Specifically, we implement a non-preemptive version of EDF-on-NoC and a Round Robin arbitration policy that is configured to be theoretically equivalent to the non-preemptive EDF-on-NoC policy.

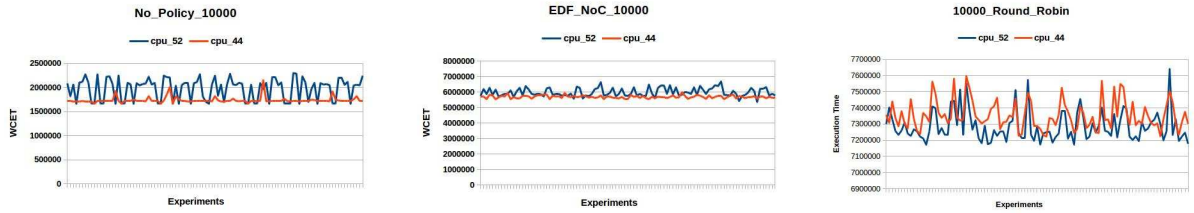
We then conduct a comparative evaluation of the performance for a user-space implementation of the variants. We compare the execution time variation of tasks on various cores. We observe that the variation ratio of execution time decreases with the use of non-preemptive EDF-on-NoC and Round Robin as memory access arbitration policies. Furthermore, we observe a consistent implementation overhead in all experiments.

[2]In further experiments, we demonstrate that this overhead is consistent for different data sizes.



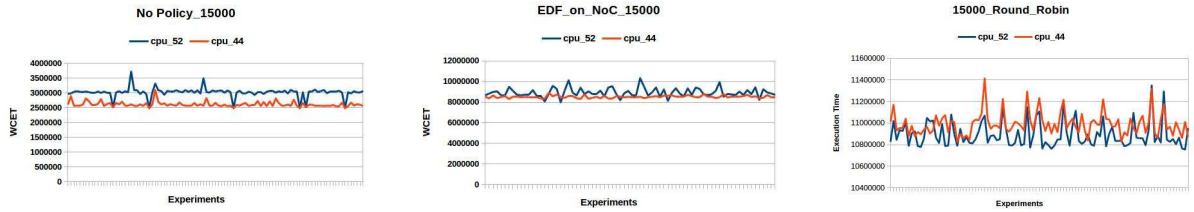
(a) No policy (b) EDF-on-NoC policy (c) Round Robin policy

Figure 4. Experiment with 5000 data set size



(a) No policy (b) EDF-on-NoC policy (c) Round Robin policy

Figure 5. Experiment with 10000 data set size



(a) No policy (b) EDF-on-NoC policy (c) Round Robin policy

Figure 6. Experiment with 15000 data set size

Our results demonstrate that, if we had an architecture like TileraPro64, but with a greater degree of control on the MDN and TDN network channels so that we can implement an off-chip memory access policy at a lower level, then task execution times on such systems could be made predictable.

REFERENCES

[1] 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*, March 2010, pp. 741–746.

[2] F. Bellosa, "Process cruise control: Throttling memory access in a soft real-time environment," Jul. 1997. [Online]. Available: http://i30www.ira.uka.de/

[3] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, July 2012, pp. 299–308.

[4] 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*, Sept 2007, pp. 251–256.

[5] M. Shekhar, H. Ramaprasad, and F. Mueller, "Network-on-chip aware scheduling of hard-real-time tasks," in *IEEE International Symposium on Industrial Embedded Systems*, 2014, pp. 141–150.

[6] "Tilera processor family," http://www.tilera.com/.

[7] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller, "Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems," in *Euromicro Conference on Real-Time Systems*, Jul. 2012.