Time-predictable Execution of Multithreaded Applications on Multicore Systems

Ahmed Alhammad
University of Waterloo, a2alhamm@uwaterloo.ca
King Saud University, ahalhammad@ksu.edu.sa

Rodolfo Pellizzoni
University of Waterloo, rpelizz@uwaterloo.ca

***Abstract*—In multicore systems, contention for access to main memory between application threads complicates timing analysis and may lead to pessimistic bounds on execution time. This is particularly problematic for real-time applications, which require provable bounds on worst-case performance. In this work, we employ a predictable execution model to schedule memory accesses performed by application threads without relying on unpredictable hardware arbiters. In addition, we statically schedule application's threads with the objective to minimize the application's makespan. Our experimental evaluation with NAS Parallel Benchmarks on 4-core system indicates that the proposed execution scheme yields an aggregated improvement of 21% over contention execution in which application's threads uncontrollably access main memory.**

I. INTRODUCTION

Modern embedded systems, including safety-critical systems, are increasingly becoming complex, interconnected systems, needing significant computation power to meet applications' demands. Since the increase in processor speed has significantly slowed down in the last decade, multicore systems are now the preferred way of supplying this increased computation demand. In this paper, we consider the scheduling of parallel, multithreaded real-time applications. Our goal is to meet the timing constraints of computation-heavy real-time applications such as synthetic vision, object tracking and structural testing [1], [2] which cannot be feasibly executed on a single core but still require guaranteed performance. At the same time, many such applications are easily parallelizable.

A large body of work in the embedded community has focused on both programming models and scheduling solutions for multithreaded applications. In particular, the fork-join parallel model has recently received significant attention in the real-time domain [3], [4]. While such works are able to provide strict guarantees on the overall response time of parallel applications, they rely on the assumption that the worst-case computation time of each individual thread can be computed independently of the application schedule. However, in practice the presence of shared architectural resources such as caches, interconnects and main memory makes such analysis extremely difficult; the execution time of concurrent threads can be significantly larger compared to the case where each thread is run in isolation. While several analysis methodologies have been proposed to compute safe bounds on the interference on shared resources (for example, see [5]), such techniques tend to be pessimistic since they must account for worst-case access patterns based on often unpredictable hardware arbiters.

To overcome such issues, we argue that it is not sufficient to focus on scheduling the cpu resource: instead, the derived application schedule should consider all physical shared resources in the system. As a first contribution in this direction, the main focus of this paper is a methodology to run a real-time multithreaded application on a multicore system with the objective to minimize the application's *makespan*, i.e., the response time of the application assuming that it runs alone on a set of m homogeneous cores. Our methodology simultaneously co-schedules the allocation of threads' memory in local caches, the accesses to main memory, and the execution of the application's threads on the available cores. Our work has the following main contributions: (1) we propose a new *m*ulti*th*readed PRedictable Execution Model (mthPREM) to schedule resource accesses of concurrent threads at a higher level without relying on hardware arbiters. (2) We describe an algorithm to map and schedule applications using the parallel fork-join model on a set of m cores. (3) We demonstrate our methodology on the OpenMP NAS Parallel Benchmark suite [6]. To the best of our knowledge, this is the first work to demonstrate real-time scheduling of fork-join applications based on realistic benchmarks rather than synthetic applications. In particular, compared to [3], [4], we extended the model to allow for synchronization among parallel threads.

The rest of the paper is organized as follows. Section II discusses related work. We introduce the system model in Section III and the scheduling algorithm in Section IV. The evaluation is presented in Section V. Finally, we provide concluding remarks in Section VI.

II. RELATED WORK

While most of the multicore scheduling work in the real-time community is applicable to sets of periodic, independent tasks, lately there has been increasing interest in the scheduling of parallel applications. Lakshmanan et al. [3] and Saifullah et al. [4] introduced a decomposition algorithm for a generalized parallel synchronous model and provided augmentation bounds under both global and partitioned EDF. Each application is divided into segments similar to the model we employ in this work, during each segment, a fixed number of threads are scheduled on the available cores. However, all existing work in real-time parallel scheduling are only concerned with the cpu resource, and do not consider interference for access to shared resources such as shared caches and main memory.

This research was supported in part by NSERC DG 402369-2011 and CMC Microsystems. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

978-3-9815370-2-4/DATE14 © 2014 EDAA

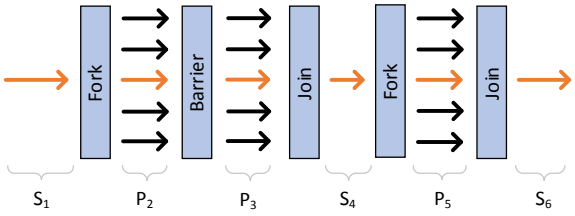

Fig. 1. Fork-join parallel model

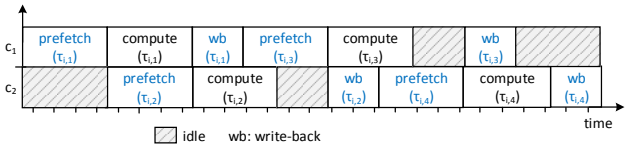

Fig. 2. Schedule example of four threads and two cores

Boniol et al. [7] propose deterministic execution models to control the access to shared resources. Similarly, Pellizzoni et al. [8] introduced a predictable execution model which co-schedules all active components in the system without relying on hardware arbiters. The model splits each task into a sequence of memory and execution phases. The key idea is to avoid cache misses by loading all required data before executing a portion of the task's code. Yao et al. [9] applied the predictable model to a multicore system under partitioned fixed-priority scheduling. Similarly to our work, the methodology in [9] is able to obtain predictable bounds on worst-case response time, but the results are not comparable since it is concerned with sets of independent periodic tasks while we focus on scheduling among threads belonging to the same application.

III. SYSTEM MODEL

We consider an application employing the *fork-join* parallel model in which the application alternates between sequential and parallel segments as in Fig. 1. The application starts as a single sequential thread then splits into multiple threads which execute concurrently. In addition, the parallel segment may be followed by multiple other parallel segments that are separated by *barrier* synchronization. After the parallel segment, all threads synchronize again into one sequential thread. This fork-join structure can be repeated multiple times. Fork-join is a popular programming model employed in systems such as OpenMP and Java [10].

We represent the parallel application as a sequence of sequential and parallel segments, $A = (S_1, P_2, P_3, \ldots, S_l)$, where l is the total number of segments. We further represent each parallel segment as a set of concurrent threads $P_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,n_i}\}$, where n_i is the total number of threads in the parallel segment P_i. The number of threads can vary between parallel segments and can also exceed the number of available cores. In the context of our work, we assume threads can access shared data in mutually exclusive way, but they have no precedence constraints within the parallel segments, i.e., there is no partial order imposed on their execution.

We are interested in scheduling a fork-join parallel application, as described above, on a processor with multiple cores $C = \{c_1, \ldots, c_m\}$, all have the same access to a single shared main memory. We assume that each core is connected to a private last-level cache with write-back policy and has a capacity that can fit both code and data of application's threads one at a time. The model can also be extended to include a shared cache that is partitioned among all cores, given that they all can concurrently access the shared cache without timing interference from each other. Furthermore, there is no hardware-implemented cache coherency protocol, assuming the cache coherency is software managed by leveraging `FLUSH` instruction to explicitly move data from private caches.

A. Predictable Execution Model

We adopt the Predictable Execution Model (PREM), first introduced in [8], to avoid contention for access to main memory from a higher level: the scheduler, without relying on unpredictable hardware arbiters. In the original PREM model, the application is sequential and divided into multiple segments. Each segment is then split into two phases: memory prefetch and computation. During the prefetch phase, all code and data that are required during the computation phase is prefetched into the core's private cache. Therefore, the core executing the segment will not suffer cache misses and will never access main memory during the computation phase.

In the fork-join parallel model, the application segments are not all sequential. Rather, the parallel segments have concurrent threads that execute on different cores. In addition, the original predictable execution model did not explicitly schedule write-back operations, instead naively assuming the data that are modified during the computation phase of one segment could be evicted from the cache and written-back by the next segment executed on that core. Thus, we extend the execution model to schedule concurrent threads on parallel segments and include a third phase for memory write-backs. During the write-back phase, we flush to main memory all cache lines modified during the preceding computation phase.

Based on the proposed execution model (mthPREM), our scheduling algorithm enforces a *contentionless memory schedule*: only one thread at a time in the entire system is allowed to run a prefetch or write-back phase; other threads can run in parallel on the remaining cores, but they must be in their computation phase. This ensures that memory phases (prefetch and write-back) will not suffer contention from other threads for access to main memory, making their running time highly predictable. Similarly, computation phases are also predictable in the sense that all memory accesses result in cache hits. Therefore, the worst-case computation time analysis for each thread can be done in isolation without having to consider the interference caused by other threads executing concurrently on other cores. In Fig. 2, we show a schedule example of a parallel segment P_i with 4 threads executing on two cores corresponding to the proposed execution model.

It is particularly important to facilitate the prefetch and write-back phases of our execution model. Thus, the platform API should provide cache prefetching primitives; otherwise, `LOAD` instructions can be used to move data from main memory. In addition, the private caches should employ a write-back policy and have primitives to invalidate some cache lines and write them to main memory. For example, Intel platforms provide the `CLFLUSH` instruction [11].

Our method relies on obtaining the set of memory blocks accessed by each thread to determine the memory blocks for

prefetch and write-back phases. We represent the application memory footprint as a set of memory blocks, including both code and data, where each block could be, for example, a cache line or memory page. A distinction between reads and writes should be captured to implement the prefetch and write-back memory phases before and after the computation phase, respectively. In general, the memory blocks can be determined by means of either compiler-driven approaches [12], using program annotations [8] or using measurement-based methods [13].

Furthermore, during the prefetch phase, self-eviction should be avoided, in a sense that loading a cache line should not evict another cache line loaded within the same phase. To illustrate how memory blocks can be allocated without conflicts, we can think about cache memory as a two dimensional array in which the columns are the cache ways and the rows are the cache sets. In systems with virtual memory, the memory blocks can be allocated without conflict along cache ways by re-arranging the physical addresses. However, in caches with non-deterministic replacement policy, only one cache way can be utilized. Fortunately, some cache controllers have a *lockdown* feature that can be implemented at the granularity of either a single line or way. This feature allows memory blocks to be hooked on a particular cache way without being overwritten until they are explicitly unlocked. In fact, the *Colored Lockdown* technique proposed in [13] can be used to avoid conflicts in prefetching thread's code or data. The key idea of this technology is to combine coloring and cache locking to deterministically control cache allocation of memory blocks. With this combined method, the whole cache can be utilized to allocate memory blocks without conflicts.

The computed schedule determines on which core each thread should execute and the start time of its memory phases. There is no need to maintain the computation phase start time because it starts immediately after the prefetch phase. There are two possible ways to implement the schedule of memory phases of each thread . First, the order of these phases can be activated on-line using a timer. In this case, the multicore system should have a common clock to synchronize the activation of memory phases across all cores. Another way of activating these phases, according to the schedule order, is by using semaphores. In this implementation, each two consecutive memory phases share a common semaphore where one waits and the other signals. This semaphore mechanism propagate through all threads memory phases to enforce the execution order commanded by the schedule.

IV. Scheduling Algorithm

In this section we discuss how to develop a static schedule of the application threads according to the model explained in Section III, and with the objective to minimize the application makespan. In general, threads scheduling involves two steps: (1) threads assignment to cores and (2) threads execution ordering. The latter step is required even though our model assumes no precedence constraints between threads within parallel segments. It is easy to see that different orderings lead to different lengths of application's makespan.

Even though individual threads are allowed to have a temporary inconsistent view of global memory within parallel segments, shared data that are protected by locks need to be made consistent immediately using `FLUSH` instruction. In fact, accessing protected shared data within a parallel segment violates the basic two assumptions of our model in that (1) no memory accesses during the computation phase and (2) memory phases do not suffer contention from other threads. It could happen that while one thread is in its prefetch phase, some other thread is in its computation phase but accesses protected shared data. Thus, we need to account for this delay for both computation and memory phases. Let the number of memory blocks that are shared for each thread be $\tau^s_{i,j}$. Then, the computation time for each thread can be obtained as:

$$\tau^c_{i,j} = \tau^{c'}_{i,j} + m \times D_m \times \tau^s_{i,j}, \quad (1)$$

where $\tau^{c'}_{i,j}$ is the computation time of each thread in isolation which can be obtained by using either static analysis or measurement-based methods. The second term in Equation 1 assumes the worst case in which all other cores are in a memory phase where D_m is the memory access time to transfer one memory block and m is the number of cores. Furthermore, the memory prefetch time is:

$$\tau^m_{i,j} = D_m \times (\tau^{urw}_{i,j} + m \times \min(\tau^{urw}_{i,j}, \sum_{\forall k \neq j} \tau^s_{i,k})), \quad (2)$$

where $\tau^{urw}_{i,j}$ is the number of *unique* memory blocks that are required by thread $\tau_{i,j}$. It includes code, read-data and write-data memory blocks. We prefetch write-data to prevent write misses during the computation phase. In opposite to computation phases, memory phases can be overlapped by only computation phases. Thus, we assume the worst case where all other cores are in computation phase and accessing protected shared data. Similarly, the memory write-back time is:

$$\tau^w_{i,j} = D_m \times (\tau^{uw}_{i,j} + m \times \min(\tau^{uw}_{i,j}, \sum_{\forall k \neq j} \tau^s_{i,k})), \quad (3)$$

where $\tau^{uw}_{i,j}$, is the number of *unique* write-data memory blocks that need to be flushed and written back to main memory.

When the number of threads, within a parallel segment, does not exceed the number of cores, scheduling involves only threads ordering. A polynomial-time algorithm that schedules these threads can be constructed. Due to space limitation, we only provide the intuition behind this algorithm. That is, threads with high computation demand should start first so that its contribution to the makespan is reduced. Another observation is that the contribution of memory prefetch phases to the makespan is the same regardless of how threads are ordered because they are serialized to prevent contention.The output of this algorithm is the start times of all memory phases for each thread. Note that the computation phase of a thread in our model starts immediately after the prefetch phase; hence, no need to maintain its start time.

On the other case where the number of threads exceeds the number of cores, scheduling involves both threads assignment to cores and threads ordering. In order to measure the quality of the schedule, we define the application *makespan* as the schedule cost function. Thus, the cost of schedule S is

$$cost(S) = \max_{c_j \in C} t_f(c_j), \quad (4)$$

where $t_f(c_j)$ is the core finish time, which is equal to the finish time of last thread scheduled on that core.

MTHPREM-STATIC shows how the schedule is constructed for each parallel segment P_j given the core allocation,

(a) Cores allocation (3 cores) (b) Threads ordering

Fig. 3. GA chromosomes

MTHPREM-STATIC

1 $t_{mem} = 0$
2 $wb(c_j) = 0, \forall c_j \in C$
3 $t_f(c_j) = 0, \forall c_j \in C$
4 Insert threads $\tau_{i,j} \in P_i$ into list L, according to the schedule order
5 **for** each $\tau_{i,j} \in L$
6 $t_m(\tau_{i,j}) = max(t_{mem}, t_f(proc(\tau_{i,j})))$
7 $t_{mem} = t_m(\tau_{i,j}) + wb(proc(\tau_{i,j})) + \tau^m_{i,j}$
8 $t_f(proc(\tau_{i,j})) = t_{mem} + \tau^c_{i,j}$
9 $wb(proc(\tau_{i,j})) = \tau^w_{i,j}$
10 Sort cores $c_j \in C$ into list L, in descending order according to their finish time
11 **for** each $c_j \in L$
12 $t_w(c_j) = max(t_{mem}, t_f(c_j))$
13 $t_{mem} = t_w(c_j) + wb(c_j)$
14 $t_f(c_j) = t_{mem}$
15 **return** $\max_{c_j \in C} t_f(c_j)$

$proc(\tau_{i,j})$ and threads execution order. To simplify the complexity of the algorithm, the memory write-back phase for each thread, except the last scheduled threads, is merged with the memory prefetch phase of the subsequent thread that is scheduled on the same core. Thus, the output of this algorithm is the start times of memory prefetch phases for all threads, namely, $t_m(\tau_{i,j})$ and the memory write-back phases for last scheduled threads on each core, $t_w(c_j)$. The **for** loop in Line 5 schedules the memory prefetch and write-back phases for all threads except the write-back phases of last scheduled threads on each core. The max in Line 6 is used to schedule the write-back phase after the computation phase. The wb in Line 7 holds the write-back phase of the previous thread, scheduled on the same core, to be merged with the prefetch phase of the current thread. The **for** loop in Line 11 schedules the write-back phases for last scheduled threads on each core. The cores are first sorted in descending order according to their finish time to reduce the contribution of write-back memory phases on the application makespan.

So far, we computed the schedule in MTHPREM-STATIC assuming that cores allocation (the spatial assignment) and threads ordering (the temporal assignment) are given. Now, we discuss the problem of optimizing such decisions. In fact, finding a schedule of minimum cost is in general difficult problem [14], because as number of threads and cores increases, the total number of possible schedules becomes vast. Consequently, it is impractical to do an exhaustive search to find the optimal solution. Stochastic search algorithms are good candidates for tackling such problems. Random search is one possible technique. In [15], the authors show that evaluating several hundred or several thousand random thread schedules is enough to get, with high confidence, close to optimal solution, given that the random samples are independent and identically distributed.

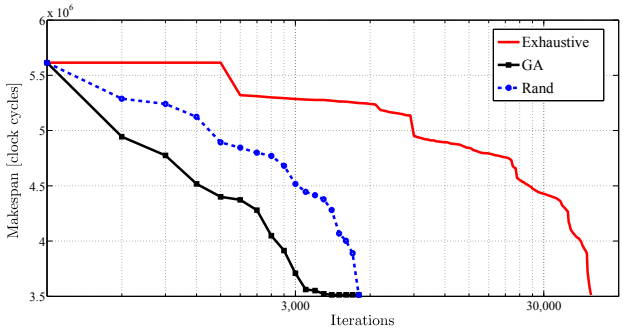

Fig. 4. Comparison between exhaustive, random and GA

Moreover, meta-heuristic search algorithms, such as genetic algorithm (GA), is another technique often used to speed up the search process. Therefore, we developed a GA algorithm to search for a good solution to our scheduling algorithm. For cores allocation, we use the value encoding scheme to represent the solution as shown in Figure 3(a). In this solution encoding (the chromosome), each thread is assigned an integer number from 1 to m, the number of cores, corresponding to its core allocation. For threads ordering, we use the permutation encoding to represent the solution as shown in Figure 3(b). In this solution encoding, each thread is assigned an integer number from 1 to n, the number of threads, corresponding to its execution order. GA outlines the major steps of our GA algorithm where G is the number of generations. We employed a single point crossover between two chromosomes for both cores allocation and threads ordering. However, threads ordering crossover has to be corrected to make sure that no two threads have the same order. For mutation operator, we randomly pick two threads, then swap their order. Figure 4 shows a comparison between GA, random and exhaustive search for 6 threads and 2 cores. Both GA and random search run for 5000 iterations, and the exhaustive runs for the whole search space which is 46080 iterations. Note that we set $G = 98$ for GA to get 5000 iterations. The GA algorithm reaches close to 5% of the optimal solution in less than 3000 iterations whereas random search reaches the same value at close to 5000 iterations.

GA

1 Create initial population of random 100 schedules
2 Evaluate the schedules as in MTHPREM-STATIC
3 **for** $i = 1 \rightarrow G$
4 Move the best 50 schedules to next generation
5 Crossover the best 50 allocation/ordering chromosomes
6 Mutate the best 50 ordering chromosomes
7 Evaluate the new 50 schedules
8 **return** the best schedule

V. EXPERIMENTAL EVALUATION

We evaluate mthPREM execution model on OpenMP NAS Parallel Benchmarks (NPB) [6]. The benchmarks consist of five parallel kernels and three simulated applications. We report results for all benchmarks except for MG because it has `ordered` OpenMP constructs which do not comply with our model in that there is no partial order imposed on threads execution within a parallel segment. These benchmarks execute sequentially until a parallel construct is encountered such

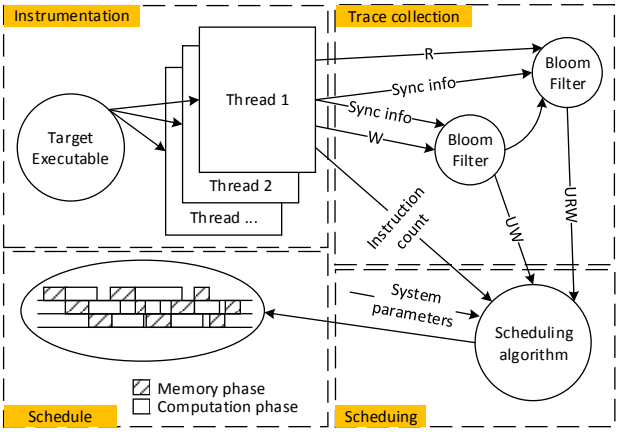

Fig. 5. Evaluation framework

as `parallel start` and `parallel end` pair. The code enclosed in the parallel construct is then executed in parallel by multiple threads. These concurrent threads synchronize their execution using `barrier` constructs. The parallel segments are separated into multiple parallel segments when barriers are encountered as explained in Section III. Moreover, shared data are accessed in mutual exclusion using `atomic` constructs. We account for these memory accesses as explained in Section IV. Table I shows the OpenMP parallel, barrier synchronization and atomic statistics for these benchmarks. The use of atomic operations is clearly low on these benchmarks; consequently, their interference to computation and memory phases is also low.

TABLE I. NPB CHARACTERISTICS

	parallel	barrier	atomic
is	16	22	0
ep	4	0	1
cg	51	2057	16
ft	42	0	6
bt	312	255	0
sp	916	415	0
lu	118	269	0

We evaluate the performance of mthPREM using the framework shown in Fig. 5. We develop a Pin-based [16] dynamic instrumentation tool to analyze the NPB benchmarks. A central element of the tool is a pervasive memory profiler to capture the memory traces for each thread. In real applications, the memory profiler is expected to capture hundreds of millions of events. Thus, we add some strategies to reduce the number of captured events that are enough to evaluate our model. First, we trace memory accesses at the granularity of cache lines which cut the size of the generated trace by 16 in the case of 64B cache lines. Second, as detailed in Section IV, the prefetch and write-back phases are mainly determined by the number of *unique* cache blocks accessed during the computation phase. In particular, both unique read and write cache blocks are captured for prefetch phase of each thread, and the unique write cache blocks for write-back phase of each thread. To capture only the unique addresses, for each thread we use a *bloom filter*. The tool also captures the mutually exclusive constructs to trace the shared memory accesses to be used by the scheduling algorithm. Furthermore, the tool captures the synchronization information of the application for each thread (e.g., `parallel` and `barrier`). This synchronization points are used to reset the counters for urw and uw, and record the trace information for each thread during a parallel segment.

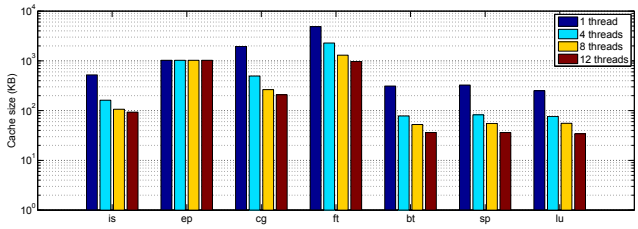

Fig. 6. Maximum data set size

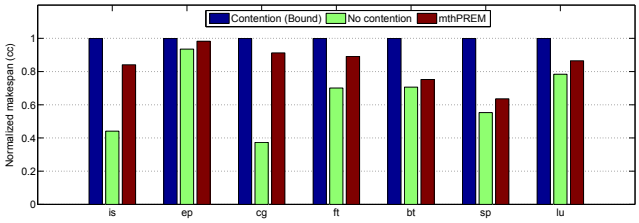

Fig. 7. Simulation results for 8 threads and 4 cores

As a system parameters, we model a simple 4-core processor with 1GHz clock frequency and realistic memory system; memory instructions that hit in cache and non-memory instructions take one clock cycle. The memory access time, D_m, for one 64B cache block is 150 clock cycles, assuming a typical DRAM with 800 MB/s bandwidth and 70ns access latency. We then apply the scheduling algorithm for each parallel segment based on the generated trace and the system parameters. We compute the total schedule cost, the application makespan, by adding the cost of all segments together. The result of the scheduling algorithm is a schedule that determines the start times of memory phases for all threads which can be activated on a multicore system using the methods we discussed in Section III.

We compare mthPREM against *contention* execution in which all threads uncontrollably access main memory. With round-robin arbitration scheme, the safest bound is to assume each memory access will suffer a delay equals to the number of cores available on the system. In fact, the work in [5] computed the bound, and concluded that the delay each thread might suffer increases linearly with the number of cores. For the sake of fair comparison, we assume, in favor of contention execution that there is no conflict cache misses by applying the same technology as in Section III. This assumption helps isolating the execution time dilation due to contention for access to main memory. In other words, threads only suffer compulsory misses in which $(\tau^r_{i,j}+\tau^w_{i,j}) \times Miss\ rate = \tau^{urw}_{i,j}$, where $\tau^r_{i,j}$ and $\tau^w_{i,j}$ are all memory accesses (not only unique) for reads and writes, respectively. Since we assume the data set of each thread fits, one at a time, within the local cache of one core, there is no capacity misses. Fig. 6 shows the maximum data set size for all benchmarks. The maximum is taken over all threads for each benchmark. It is clear that data set sizes decrease as number of threads increases. That means, with a fixed size of local cache, changing the number of threads allows some applications to execute according to mthPREM. Note that for some benchmarks like EP, the maximum data set size remains constant. We argue that the reason is due to the type of parallelism within the parallel segment. For work sharing constructs such as `parallel for`, the data is distributed and processed collaboratively by all threads. Therefore, the amount of data each thread

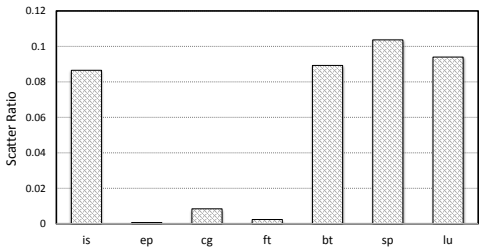

Fig. 8. Cache lines scatter ratio

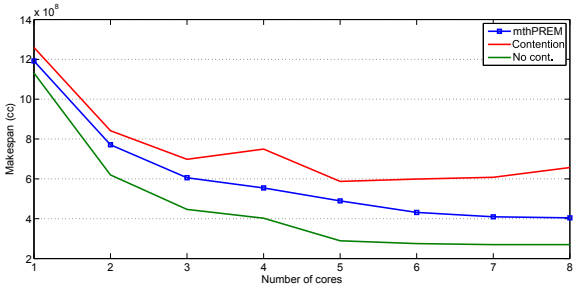

Fig. 9. mthPREM scalability by varying the number of cores

need to process reduces as number of threads increases. In contrast, constructs like `sections`, implement different type of parallelism known as task parallelism in which threads concurrently execute independent codes. Hence, the amount of data processed remains constant, but the code size for each thread changes.

Fig. 7 shows a simulation results for NAS Parallel Benchmarks, all with class S data set, 8 threads and 4 cores. We get these results by running the GA algorithm for 5000 iterations. The *no contention* bar is the best case execution where memory accesses do not suffer any delay from other threads. The no contention execution is really optimistic, and we report it only to see the lower bound of our proposed model. The mthPREM model achieves an aggregated improvement of 21% over the worst case bound. The benchmarks show different improvements because they have different ratios of memory to computation. As this ratio increases, the improvement of our method also increases.

As we previously mentioned, the prefetch memory blocks need to be touched by inserting loops at the beginning to bring the data into the cache. With unit-stride memory blocks, only one address pointer is sufficient to load all required data. In contrast, non-stride memory blocks need a data structure to keep track of all memory blocks. This clearly adds an overhead to the prefetch phase. Note that the same applies for write-back phases. Indeed, the overhead depends on how scattered is the layout of memory blocks in main memory. We cluster the data set into chunks of contiguous memory blocks, and we define the scatter ratio as basically the ratio between the number of chunks and the total number memory blocks. In other words, 0% scatter ratio means continuous layout of all memory blocks. Fig. 8 shows the prefetch overhead for all benchmarks. In some benchmarks, almost all memory blocks are consecutive, and for others, there is some scattering but limited by less than 10%.

Finally, Fig. 9 shows the scalability of mthPREM as number of cores increases. This plot is for LU benchmark with 16 threads. The application's makespan difference between mthPREM and conention execution is continuously increasing. Indeed, increasing the number of cores is advantageous for our method compared to contention execution because the amount of contention in memory becomes large with increasing number of cores. In contrast, mthPREM avoids contention in memory regardless of the number of cores.

VI. CONCLUSION

In this work, we proposed a new methodology to predictably schedule real-time multithreaded applications on multicore systems. The proposed solution showed a significant improvement on the execution performance as a result of avoiding the contention between threads for access to main memory. We plan to extend our work to co-schedule multiple applications with the proposed scheme.

REFERENCES

[1] H.-M. Huang, T. Tidwell, C. Gill, C. Lu, X. Gao, and S. Dyke, "Cyber-physical systems for real-time hybrid structural testing: a case study," in *Proc. Int. Conf. Cyber-Physical Systems*. ACM, 2010, pp. 69–78.

[2] A. Kurdila, M. Nechyba, R. Prazenica, W. Dahmen, P. Binev, R. DeVore, and R. Sharpley, "Vision-based control of micro-air-vehicles: Progress and problems in estimation," in *Proc. Decision and Control*, vol. 2. IEEE, 2004, pp. 1635–1642.

[3] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proc. RTSS*. IEEE, 2010, pp. 259–268.

[4] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, "Multi-core real-time scheduling for generalized parallel task models," in *Proc. RTSS*. IEEE, 2011, pp. 217–226.

[5] R. Pellizzoni, A. Schranzhofery, J.-J. Cheny, M. Caccamo, and L. Thieley, "Worst case delay analysis for memory interference in multicore systems," in *Proc. DATE*. IEEE, 2010, pp. 741–746.

[6] H. Jin, M. Frumkin, and J. Yan, "The openmp implementation of nas parallel benchmarks and its performance," Technical Report NAS-99-011, NASA Ames Research Center, Tech. Rep., 1999.

[7] F. Boniol, H. Cassé, E. Noulard, and C. Pagetti, "Deterministic execution model on COTS hardware," *Architecture of Computing*, pp. 98–110, 2012.

[8] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *Proc. RTAS*. IEEE, 2011, pp. 269–279.

[9] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, 2012.

[10] D. Lea, "A java fork/join framework," in *Proc. Conf. Java Grande*. ACM, 2000, pp. 36–43.

[11] *IA-32 Architectures Software Developers Manual*, Volume 3[a] ed., Intel, 2011.

[12] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 2, pp. 472–511, May 2006.

[13] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *Proc. RTAS*. IEEE, 2013, pp. 45–54.

[14] A. Darte, Y. P. Robert, F. Vivien, and F. Vivien, *Scheduling and automatic Parallelization*. Springer, 2000.

[15] P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F. Cazorla, M. Nemirovsky, and M. Valero, "Optimal task assignment in multithreaded processors: a statistical approach," in *Proc. ASPLOS*, 2012, pp. 235–248.

[16] M. Bach, M. Charney, R. Cohn, E. Demikhovsky, T. Devor, K. Hazelwood, A. Jaleel, C.-K. Luk, G. Lyons, H. Patil *et al.*, "Analyzing parallel programs with pin," *Computer*, vol. 43, no. 3, pp. 34–41, 2010.