On Cache-Aware Task Partitioning for Multicore Embedded Real-Time Systems

Aaron Lindsay
CS Department
Virginia Tech
Blacksburg, VA 24061
Email: aaron@aclindsay.com

Binoy Ravindran
ECE Department
Virginia Tech
Blacksburg, VA 24061
Email: binoy@vt.edu

Abstract

One approach for real-time scheduling on multicore platforms involves task partitioning, which statically assigns tasks to cores, enabling subsequent core-local scheduling. No past partitioning schemes explicitly consider cache effects. We present a partitioning scheme called LWFG, which minimizes cache misses by partitioning tasks that share memory onto the same core and by evenly distributing the total working set size across cores. Our implementation reveals that LWFG improves execution efficiency and reduces mean maximum tardiness over past works by as much as 15% and 60%, respectively.

1. Introduction

Most multicore processors manufactured today use a hierarchical cache scheme. Cache misses have the potential to be more costly as the memory hierarchy deepens, and should be avoided if at all possible. Furthermore, as the core counts increase and applications adapt to take advantage, the possibility of cross-processor memory interference likewise increases. This interference manifests itself when multiple processors access and update the memory from the same region within a short period of time, causing cache-invalidations and subsequent cache misses.

Hard real-time systems presume knowledge of worst-case execution times (WCETs). Unfortunately, calculating WCETs is difficult for modern multicore hardware [1], largely due to the presence of shared caches, as this prevents task execution behaviors from being analyzed in isolation [2]. While WCET research for multicores is actively progressing [3], for many current non safety-critical real-time systems, a tight, measurement-based execution time estimate is often sufficient (e.g., [4]). Such an approach is cost-effective, given the current lack of techniques and tools for determining WCETs for arbitrary program/processor pairs. In these situations, decreasing cache misses can help minimize the potential under- or over-estimation of the execution time with respect to the actual execution time, thereby improving task timeliness behavior (e.g., improved instructions per cycle, reduced tardiness). Moreover, embedded real-time systems may also benefit from fewer cache misses by exploiting the extra slack in task schedules to reduce power consumption.

The real-time multiprocessor scheduling space can be divided into: global, clustered, and partitioned. In global scheduling, all tasks are scheduled at each scheduling event, potentially migrating them across processing cores. While there exist theoretically optimal global schedulers (e.g., [5]), studies have shown their limited scalability [6].

In partitioned scheduling, tasks are statically divided between cores, for subsequent core-local scheduling, thereby converting a m-core scheduling problem into m uniprocessor scheduling problems. The EDF algorithm can feasibly schedule a taskset if the sum of task densities is not greater than one. Therefore, the only concern is whether or not a taskset can be feasibly partitioned.

The partitioned approach is attractive due to its relative simplicity compared to global scheduling and the elimination of costly migrations. Unfortunately, the task partitioning problem is analogous to the bin-packing problem and is NP-hard [7]. If the sum of all task densities in a taskset approaches the number of cores, it becomes difficult to find a feasible partitioning such that each core has tasks with a total density of less than one (see Section 2 for the definition of task

density).

Finally, there are algorithms that compromise between global and partitioned scheduling. They range from semi-partitioned [8], where most tasks are partitioned but a few are designated as 'migratory', to clustered [9], where tasks are partitioned onto groups of cores.

Partitioned, semi-partitioned, and clustered algorithms may be considered somewhat cache-aware, as they restrict task migrations. Restricting migrations makes it more likely for a task's working set to reside at least partially in the cache when it resumes execution, whereas a task migrating to a processor that shares no caches with the source processor would be forced to fetch its entire working set from main memory. By limiting the maximum cache distance a task is allowed to migrate, these approaches decrease the potential for costly cache misses.

To our knowledge, no past work has attempted to $partition$ tasks with the goal of minimizing cache misses and improving execution efficiency. The main class of algorithms which consider the memory hierarchy are global algorithms that encourage or discourage the co-scheduling of tasks which share memory [10], [11]. The only work that we are aware of that deals with cache-aware clustered real-time scheduling is [12], which describes a clustered scheduling algorithm for soft real-time applications. However, [12] is not applicable to hard real-time systems; the paper does not clearly describe the algorithm implementation, and only presents a limited simulation experiment.

Our contributions are: (1) We characterize how a partitioning strategy may affect the number of cache-misses. (2) We design a cache-aware partitioning scheme that seeks to minimize cache misses, while satisfying hard real-time requirements. (3) We implement our partitioning scheme in a real-time Linux kernel and evaluate it on a 48-core hardware platform, and show that it increases execution efficiency by as much as 15% over cache-unaware partitioning algorithms. Interestingly, mean maximum tardiness is also reduced by up to 60%.

2. Task Model and Preliminaries

We assume [7]'s task model: each task τ_i has a WCET e_i, a minimum task-arrival separation p_i, and a deadline d_i. The density of a task τ_i, $\lambda_i = e_i/min(p_i, d_i)$. When $d_i \leq p_i$, τ_i's density is equal to its utilization, $u_i = e_i/p_i$.

We assume that the maximum working set size (WSS) for each task is known. This is a reasonable assumption, because knowledge of a task's WSS is required for any accurate WCET calculation which takes cache misses into account. The maximum WSS for a task τ_i is denoted w_i. Our partitioning scheme relies upon the fact that a significant portion of a task's working set remains the same over a period of time larger than the frequency of scheduling events and context switches. (Exactly how long the WSS must remain the same for our algorithm to have a positive effect is future research.) Additionally, we rely on knowing which tasks share memory and how much (though for the sake of simplicity, we assume tasks which share memory share their entire WSS).

We define the *cache distance* between two cores A and B as the best-case memory access latency from B to a word of memory that resides (and has not been invalidated) in A's L1 cache.

One problem with existing partitioning approaches is that it is possible for the tasks with the largest WSSes to be distributed unevenly among the available processors. If the tasks partitioned on a particular core have a WSS greater than the size of the largest cache (the last level cache, or LLC), it is impossible for their collective memory to be continuously housed in the LLC over the lifetime of the application. As the collective per-core WSS approaches and passes the size of the LLC, the number of cache misses per task execution is likely to increase.

Furthermore, it is advantageous to schedule tasks that share memory with each other on either the same core or on cores which are a small cache distance apart, for two reasons. First, scheduling them on 'cache-close' cores increases the likelihood that the previously-running task may share memory with the current task, keeping the cache warmer and decreasing the number of cache misses. Second, scheduling memory-sharing tasks on different cores will result in a cache-invalidation whenever they modify memory which shares the same cache line as memory currently residing in the other's cache. Cache invalidations both cause more traffic on the shared bus and force the core whose cache line was invalidated to acquire a new copy of that memory whenever it is accessed again. Avoiding such contention is therefore also a priority.

3. Cache-Aware Partitioning

Building on our previous observations, we formulate two goals for our cache-aware partitioning scheme above those for traditional partitioning: (1) Evenly distribute the collective WSS of a taskset over all cores to decrease conflict and capacity-related cache misses. (2) Partition tasks that share memory so that, the sum

of the cache distance between all such pairs of tasks is minimized.

3.1. Existing Heuristics

Because bin-packing, and therefore partitioning, is NP-hard, several polynomial-time heuristics have been developed. The simplest algorithms include best-fit, worst-fit, first-fit, and next-fit [13]. In the context of partitioning, these heuristics consider tasks one-by-one in some order, and place the task under consideration onto a particular core based on that heuristic's criteria. When partitioning, we assume m bins, each with a capacity of one, where the 'size' of a task is determined by its density.

The best-fit heuristic places a task in the bin with the smallest remaining capacity that can still accommodate it. The worst-fit heuristic places a task in the bin with the largest remaining capacity. The first-fit heuristic places a task in the first bin it encounters with sufficient capacity, starting with the first bin. Finally, the next-fit heuristic places a task in the first bin it encounters with sufficient capacity, starting with the bin immediately following the bin in which the last task was placed.

For each of these heuristics, there are a myriad of different orders in which they may consider the tasks. There have been many variations, but the most common are non-decreasing relative deadline order [7] (i.e. $\forall i \in [0, n) : d_i \leq d_{i+1}$) and non-increasing utilization order [6] ($\forall i \in [0, n) : u_i \geq u_{i+1}$).

3.2. Largest WSS First Algorithm

To evenly distribute WSS across cores, we partition tasks with a next-fit heuristic in non-increasing WSS order (i.e. $\forall i \in [0, n) : w_i \geq w_{i+1}$). This ensures that each core has a task with one of the m largest WSSes in the taskset before adding a second task to any core.

To schedule tasks which share memory on cache-close cores, we modify our algorithm to group tasks which share memory. Tasks are still considered in non-increasing WSS order, but whenever we consider a task, we also consider all tasks that share memory with our current task that have not previously been partitioned. The algorithm attempts to schedule this group of tasks as a whole – the sum of their task densities is considered when determining if a core has sufficient remaining capacity to schedule the group. If the next-fit heuristic fails to find a core with sufficient capacity to partition the group, the task that shares the least memory with the first task is removed from the group, and partitioning is retried with the smaller group. Pseudocode for this algorithm, called LWFG, for 'Largest WSS First, Grouping', is given in Algorithm 1.

Algorithm 1 LWFG Algorithm

```
sort_decreasing_wss(taskset)
last ← 0
for all task ∈ taskset do      ▷ Find group members which share memory with task
    group ← get_memory_sharers(taskset, task)
    partitioned ← False
    while partitioned == False do
        for all cpu ∈ cpus start after last do      ▷ Loop through cpus starting after last assigned
            if capacity_left(cpu) ≥ density(group) then
                partition(group, cpu)
                last ← cpu
                partitioned ← True
                break
            end if
        end for
        if num_tasks(group) > 1 then      ▷ Remove task from group if possible
            remove_least_shared(group)
        else
            return Partitioning Failed
        end if
    end while
    taskset.remove(group)
end for
return Partitioning Succeeded
```

Complexity. With m processors and n tasks, LWFG's worst-case complexity is $O(n^2 m)$. Though this is costlier than the simple *-fit algorithms by a factor of n, we were able to partition ≈50,000 tasksets using a Python implementation in less than 5 minutes on a 1.6Ghz processor.

Feasibility. Uniprocessor EDF can schedule any taskset T for which the sum of the task densities is no greater than one: $\sum_{\tau_i \in T} \lambda_i \leq 1$. Let π_j denote processor j and $P(\pi_j)$ denote all tasks partitioned to π_j. If T can be partitioned using LWFG such that the total task density on any one core is less than one (i.e. $\forall j \in [0, m) : \sum_{t_i \in P(\pi_j)} \lambda_i \leq 1$), T is schedulable on uniprocessor EDF using LWFG.

We do not attempt to establish a feasibility test for LWFG. Instead, we suggest that systems which require hard real-time schedulability 'fall back' to other partitioning algorithms (or perhaps even a global algorithm, if necessary) if LWFG fails to partition a taskset. This approach will improve performance when possible using LWFG, while still making schedulability guarantees.

4. Experimental Evaluation

We compared LWFG with the three most widely studied partitioning schemes: worst-fit in non-increasing task utilization order (WFD) [6], first-fit in

Table 1. Taskset Distribution Parameter Bounds

Taskset	Tasks/MTT	Utilization	u_i is per-	Period (ms)	p_i is per-	WSS (kB)
MLU	$[1, 4]$	$[0.01, 0.1]$	MTT	$[24, 240]$	MTT	$e_i/3 * 128$
MMU	$[1, 4]$	$[0.1, 0.4]$	MTT	$[24, 240]$	MTT	$e_i/3 * 128$
MWL	$[1, 8]$	$[0.1, 0.4]$	MTT	$[10, 250]$	MTT	$[64, 512]$
MWH	$[1, 8]$	$[0.1, 0.4]$	MTT	$[10, 250]$	MTT	$[4096, 8192]$
MWLP	$[1, 8]$	$[0.1, 0.4]$	task	$[10, 250]$	MTT	$[64, 512]$
MWHP	$[1, 8]$	$[0.1, 0.4]$	task	$[10, 250]$	MTT	$[4096, 8192]$
MWLU	$[1, 8]$	$[0.1, 0.4]$	MTT	$[10, 250]$	task	$[64, 512]$
MWHU	$[1, 8]$	$[0.1, 0.4]$	MTT	$[10, 250]$	task	$[4096, 8192]$

non-increasing task utilization order (FFD) [14], and Baruah and Fisher's first-fit in non-decreasing relative deadline order (BF) [7].

4.1. Methodology

We implemented and tested LWFG on *ChronOS Linux* [15], a real-time Linux kernel. The code for our kernel, userspace utilities, and testing infrastructure is available online at `chronoslinux.org`. Similarly to the approach in [16], [10], we randomly generated synthetic tasksets based on certain allowable parameter ranges. All tasksets were first partitioned offline before being scheduled with one instance of uniprocessor EDF per core. We used the Linux `perf` tool to measure information from the performance monitoring units (PMUs) of the target machine, including the number of LLC loads and misses, the number of instructions executed, and the number of cycles that execution took.

4.1.1. Workload. In our evaluation, each task accesses its memory in a sequential pattern, implemented by incrementing elements in an array in strides equal to the cache line size (64 bytes on our platform). We believe this memory access pattern is generic enough to be representative of other memory-intensive real-time applications. For example, the FFT algorithm largely accesses an array in-order, doing computation on each element in varying strides, depending on which pass of the algorithm it is on. Many other DSP applications also have largely in-order memory access patterns, including multimedia encoding and decoding.

4.1.2. Taskset Distributions. All our randomly-generated tasksets were generated using the multi-threaded task (MTT) model of [10]. Each MTT consists of several individual tasks that share memory with each other. Tasks within an MTT share the entirety of their WSS with one another, and no tasks not in the same MTT share memory (with the exception of the application and shared library text, which is minimal).

To generate each MTT, the number of threads the MTT will contain is first chosen randomly within the specified bounds. Next, the period and utilization for the MTT are randomly generated, from which the execution time is calculated. Finally, the WSS is either randomly generated or calculated based off of the execution time for the MTT, depending on the taskset distribution. Note that for those tasksets that allow periods or utilizations to be different between tasks in the same MTT, the previous steps are per-task rather than per-MTT.

Table 1 describes the distributions used. The first two taskset distributions are slightly modified from [10], while the remaining are of our own design. For these, we aim to test conditions not covered by previous taskset distributions – namely WSSes which vary independently of execution times, and MTTs which share memory but do not necessarily share periods or utilizations.

We exclude taskset distributions which contain tasks with large utilizations. As the average task utilization in a taskset approaches and exceeds $\frac{1}{2}$, the LWFG's benefits will diminish. This is because, if all task utilizations are greater than $\frac{1}{2}$, no two tasks can feasibly be placed on the same core, which results in a much smaller degree of cache-sharing than is possible if more than one memory-sharing task reside on the same core. We do not present results with tasksets that contain tasks with utilizations greater than $\frac{1}{2}$ because LWFG will deteriorate to other algorithms under these conditions.

4.2. Hardware Platform

Our platform is a machine with 4×12-core AMD Opteron processors (see Table 2). We tested tasksets with total utilization caps up to 48 (max number of cores).

Table 2. 12-core AMD Opteron™ 6164 HE Memory Hierarchy

Level	Shared Between	Size	Associativity
L1-I	1 core	64 Kb	2-way
L1-D	1 core	64 Kb	2-way
L2	1 core	512 Kb	16-way
L3	6 cores	5118 Kb	48-way
NUMA Node	12 cores	4 Gb	n/a
Main Memory	48 cores	16 Gb	n/a

5. Results

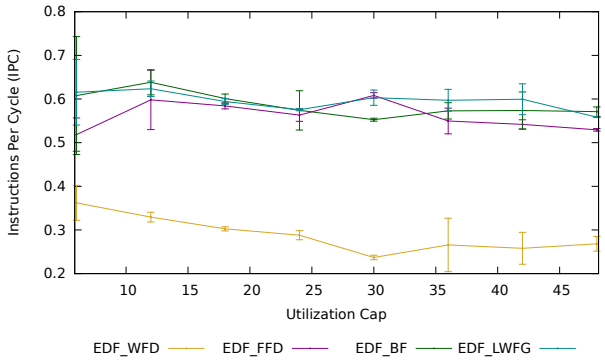
Figures 1 and 2 show the instructions per cycle (IPC) of each taskset distribution at each load. An improvement in IPC shows that a partitioning scheme not only decreases the total number of cache misses, but that this decrease causes more efficient execution overall.

Observation 1 *LWFG more than doubles the IPC of WFD in some cases.* WFD is widely studied (e.g., [6]) and is shown to be effective on evenly distributing the total taskset utilization among all available cores [14]. LWFG outperforms WFD for all distributions and all loads, and rarely shows an IPC improvement of less than 20%.

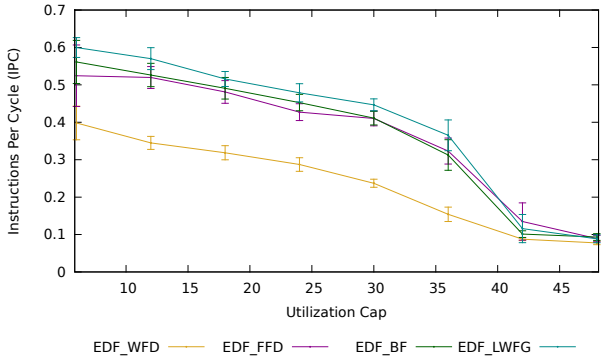
Observation 2 *LWFG's IPC is more consistent than other algorithms.* Although LWFG does not obtain significantly higher IPC than all competitors in all cases, it never performs significantly worse than the best algorithm for any distribution at any load. This shows that while competitors perform on par with LWFG for some taskset distributions, they are not as consistent as LWFG is.

For example, both FFD and BF perform nearly as well as LWFG for the MWL and MWH distributions. In these distributions, both periods and utilizations are the same for all tasks within an MTT. Because FFD and BF consider tasks in the order of task utilizations or periods when partitioning, they 'accidentally' partition most tasks in an MTT to the same core, similarly to LWFG. The performance gains by LWFG over the other algorithms are due to the fact that LWFG more evenly distributes the working set over available cores and is intentional about grouping the tasks with the largest WSSes together first. In contrast to the MWL and MWH results, we see that FFD does not perform nearly as well on the MWHP and MWLP taskset distributions. This is because, tasks within an MTT in one of these distributions do not share the same utilizations. Therefore, partitioning tasks in the order of decreasing utilization is much less likely to 'accidentally' result in a cache-aware partitioning.

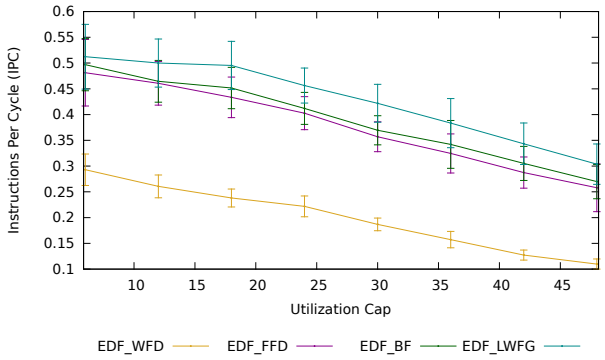
Observation 3 *LWFG does not out-perform FFD*



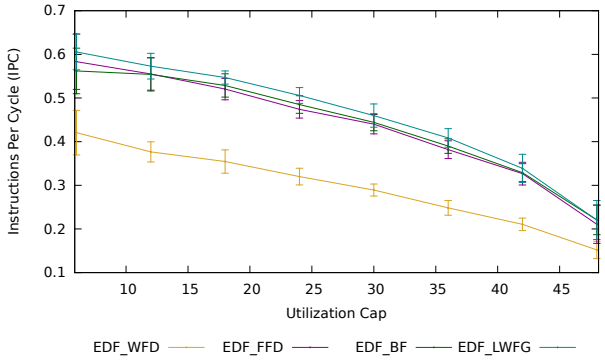
(a) IPC for MLU Distribution



(b) IPC for MMU Distribution

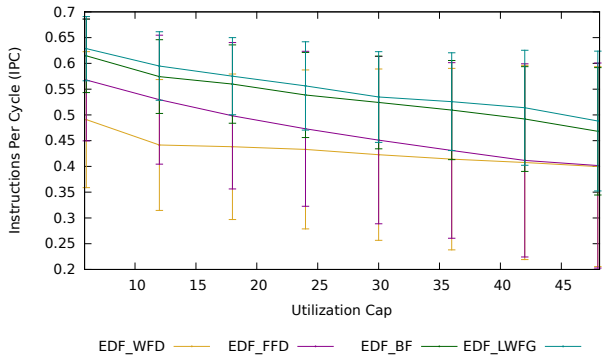


(c) IPC for MWL Distribution

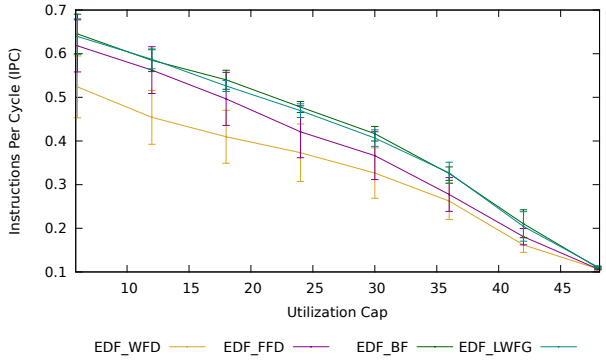


(d) IPC for MWH Distribution

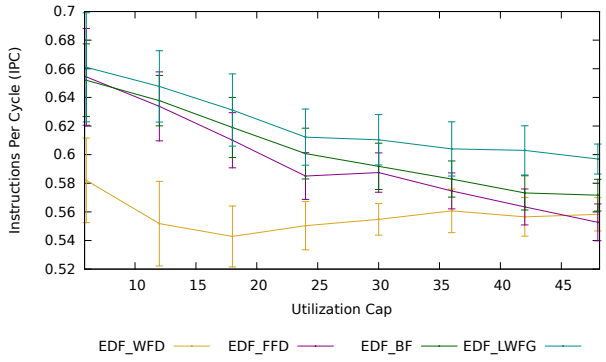
Figure 1. Instructions Per Cycle Results: MLU, MMU, MWL, and MWH Distributions.



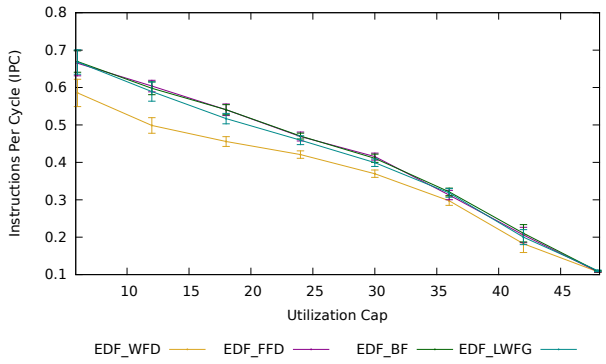
(a) IPC for MWLP Distribution



(b) IPC for MWHP Distribution



(c) IPC for MWLU Distribution



(d) IPC for MWHU Distribution

Figure 2. Instructions Per Cycle Results: MWLP, MWHP, MWLU, and MWHU Distributions.

and BF in all cases. We note that for those taskset distributions where the periods of tasks in a MTT are not all the same, but are randomly-generated on an individual basis (MWLU, MWHU), LWFG's performance drops somewhat relative to the competitors, and the IPC of LWFG is even less than that of FFD and BF on some loads. This decrease in IPC improvement is at least partially due to the timing of the task periods and deadlines. Two tasks which share the same period and are partitioned on the same core will tend to be scheduled immediately after one another in uniprocessor EDF (assuming no task with an earlier deadline arrives in the meantime). The same two tasks, if partitioned to different cores, will tend to execute at the same time if the other tasks partitioned to their processors have similar periods and deadlines relative to their own. This means that, tasksets where tasks in MTTs share periods demonstrate worst-case behavior for partitioning algorithms which do not group memory-sharing tasks. This is because, the timing of these tasks' executions will tend to maximize cache invalidations.

Relaxing the restriction that all tasks in an MTT must have the same period means that, tasks in an MTT will not tend to execute at the same time as frequently. The cache effects due to cache invalidations in this case will not be as severe as they would be if all periods remained the same in an MTT. The difference between the collective partitioning algorithms' performance for the MWHP (periods equal in MTT) and MWHU (periods randomized) distributions can be seen in Figures 2(b) and 2(d), respectively. WFD performs ≈15% better at low loads when periods are randomized, and that the IPC of all algorithms is clumped much closer together in this case.

LWFG's lack of improvement therefore is due to the naturally-better performance of the other algorithms rather than a fault of its own. Regardless, it is important to note that LWFG does not improve IPC in all cases.

Observation 4 *IPC performance in general is unpredictable on MWLP and MWHP distributions.* The error bars on the plots represent the standard deviation of all the tasksets at that point for that algorithm. It is obvious that the MWLP and MWHP distributions have much higher variation than others, particularly for those algorithms that partition tasks in order of decreasing utilization (FFD and BF). We attribute this to the fact that the utilizations of tasks in each MTT are not the same for these two distributions. Therefore, the efficacy of these algorithms is based on the random chance of two tasks in the same MTT receiving the same (or different) utilizations, rather than performing universally good or bad because all tasks in an MTT

shared the same utilization.

Figure 3 shows the deadline satisfaction ratio (DSR). DSR is calculated as the total number of deadlines met over the total number of task instances. These plots are included to demonstrate that we do not sacrifice meeting deadlines for increased efficiency and IPC. The maximum value for DSR is therefore 1.0. We omit the DSR plots for the MWLP, MWLU, MWHP, and MWHU distributions because they do not show any trends not evident on those of the MWL and MWH distributions.

Observation 5 *LWFG's DSR is generally comparable to the others.* While LWFG's DSR dips below 1.0 at high loads, it is never considerably worse than the others. This is due to infeasibly-partitioned tasksets. If a 'fall-back' partitioning approach were taken, where another scheme were used if LWFG failed to feasible partition a taskset, these deadline misses would likely be minimized.

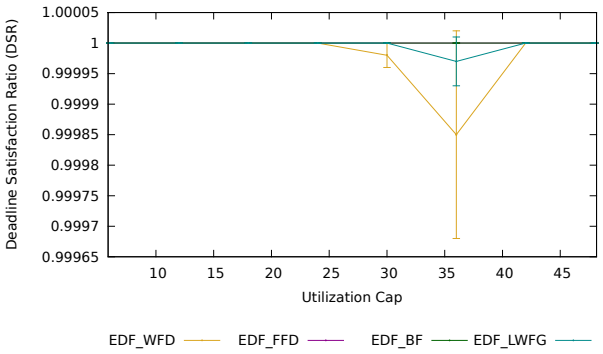
Observation 6 *BF's DSR outperforms all algorithms for tasksets with large amounts of memory.* LWFG's DSR is consistently higher than that of WFD, and is generally comparable to that of FFD. However, BF appears to be the best. While LWFG's improvement in execution efficiency helps it avoid some missed deadlines, it is not enough to overcome its higher number of infeasible tasksets. Again, this leads us to suggest using LWFG as a primary partitioning scheme, and using a 'fallback' scheme when LWFG fails to feasibly partition a taskset.

Though Figures 1 and 2 show that LWFG improves IPC, they do not reveal the impact on task timeliness because majority of the tasksets are feasible. To understand the impact on task timeliness, we varied the ratio of the actual WCET to the WCET reported to the scheduling algorithm from 1.0 to 2.0, and measured the deadline satisfaction ratio (DSR) and maximum per-taskset task tardiness. (DSR is the fraction of deadlines met.) Unlike before, the maximum taskset utilization is fixed at 95% of the theoretical bound. For brevity, we focus on the MWL distribution.

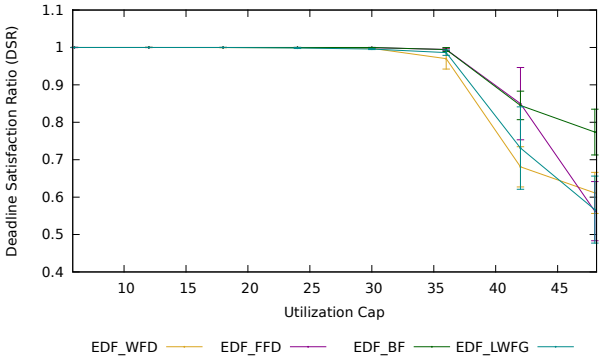
Figure 4 shows the results. While BF manages to obtain slightly higher DSR, LWFG does a much better job of bounding tardiness (up to 60% reduction over BF in Figure 4(b)). This is because, LWFG misses deadlines by smaller time magnitudes than BF. Figure 4(c) demonstrates the higher frequency of tasks with low-maximum-tardiness with LWFG than with BF.

6. Conclusions

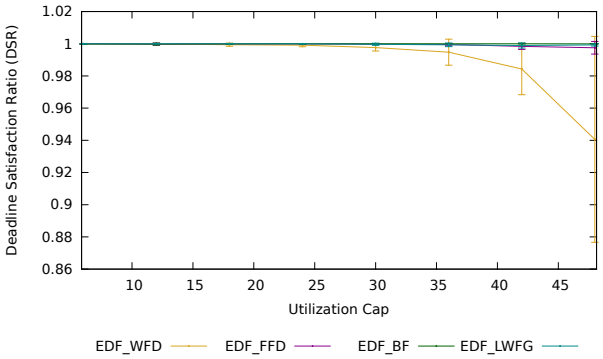
We have presented the rationale behind, and the design of, a new cache-aware, real-time task partition-



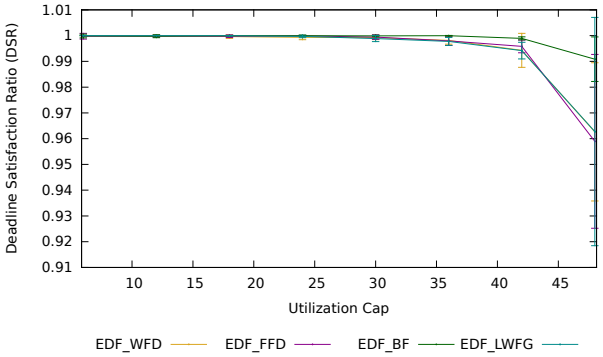
(a) DSR for MLU Distribution



(b) DSR for MMU Distribution

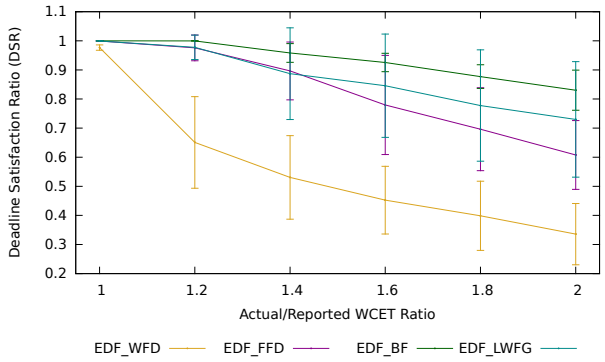


(c) DSR for MWL Distribution

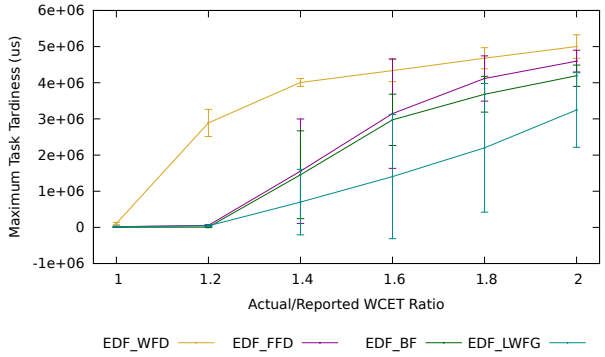


(d) DSR for MWH Distribution

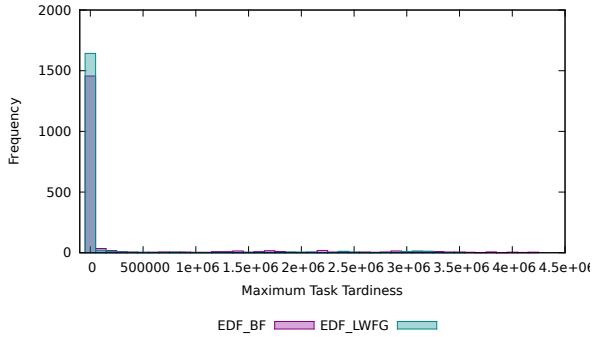
Figure 3. Deadline Satisfaction Ratio Results.



(a) DSR for MWL Distribution



(b) Tardiness for MWL Distribution



(c) Tardiness Histogram for MWL Distribution

Figure 4. Deadline Satisfactions and Task Tardiness Results

ing scheme, called LWFG. Our empirical evaluation reveals that, LWFG is effective at minimizing cache misses. It is particularly effective at improving execution efficiency in some cases (i.e., where all periods are the same within an MTT), and increases IPC by as much as 15%. It also reduces mean maximum tardiness by as much as 60%. However, there are also a few distributions for which LWFG does not significantly improve performance.

Acknowledgments

This work was sponsored by US NSWC Dahlgren under grants N00178-09-D-3017-0018/0022.

References

[1] D. Dasari, V. Nelis, and B. Andersson, "WCET analysis considering contention on memory bus in COTS-based multicores," in *ETFA*, 2011, pp. 1 –4.

[2] N. Guan *et al.*, "Cache-aware scheduling and analysis for multicores," in *EMSOFT*, 2009, pp. 245–254.

[3] ——, "WCET analysis with MRU cache: Challenging LRU for predictability," in *RTAS*, 2012.

[4] W. Yuan and K. Nahrstedt, "Energy-efficient CPU scheduling for multimedia applications," *ACM TOCS*, vol. 24, no. 3, pp. 292–331, 2006.

[5] H. Cho *et al.*, "An optimal real-time scheduling algorithm for multiprocessors," in *RTSS*, 2006, pp. 101–110.

[6] A. Bastoni *et al.*, "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers," in *RTSS*, 2010, pp. 14 –24.

[7] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *RTSS*, 2005, pp. 321–329.

[8] S. Kato *et al.*, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *ECRTS*, 2009, pp. 249 –258.

[9] J. Calandrino, J. Anderson, and D. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *ECRTS*, 2007, pp. 247 –258.

[10] J. M. Calandrino and J. H. Anderson, "On the design and implementation of a cache-aware multicore real-time scheduler," in *ECRTS*, 2009, pp. 194–204.

[11] J. H. Anderson *et al.*, "Real-time scheduling on multicore platforms," in *RTAS*, 2006, pp. 179–190.

[12] Y. Wang *et al.*, "A shared cache-aware hybrid real-time scheduling on multicore platform with hierarchical cache," in *PAAP*, 2011, pp. 208 –212.

[13] E. G. Coffman, Jr. *et al.*, "Approximation algorithms for bin packing: A survey," in *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, 1997.

[14] H. Aydin and Q. Yang, "Energy-aware partitioning for multiprocessor real-time systems," in *IPDPS*, 2003, p. 9 pp.

[15] M. Dellinger, P. Garyali, and B. Ravindran, "ChronOS Linux: A best-effort real-time multiprocessor Linux kernel," in *DAC*, June 2011, pp. 474–479.

[16] T. Baker, "A comparison of global and partitioned EDF schedulability tests for multiprocessors." Florida State University, Tech. Rep. TR-051101, 2005.