A Hybrid Real-Time Scheduling Approach for Large-Scale Multicore Platforms *

John M. Calandrino[1], James H. Anderson[1], and Dan P. Baumberger[2]
[1]Department of Computer Science, The University of North Carolina at Chapel Hill
[2]Systems Technology Lab, Intel Corporation, Hillsboro, OR

Abstract

We propose a hybrid approach for scheduling real-time tasks on large-scale multicore platforms with hierarchical shared caches. In this approach, a multicore platform is partitioned into clusters*. Tasks are statically assigned to these clusters, and scheduled within each cluster using the preemptive global EDF scheduling algorithm. We show that this hybrid of partitioning and global scheduling performs better on large-scale platforms than either approach alone. We also determine the appropriate cluster size to achieve the best performance possible, given the characteristics of the task set to be supported.*

1 Introduction

Multicore architectures, which include several processors on a single chip, are being widely touted as a solution to the "thermal roadblock" imposed by single-core designs. Most chip makers have released dual-core chips, and a few designs with more than two cores have been released as well. For instance, both Intel and AMD have released four-core chips, Sun recently released its eight-core Niagara chip, and Intel is expected to release chips with 80 cores within five years [6]. Azul, a company that builds Java appliances, has created 48-core chips that are used in systems with up to 768 total cores [1]. These appliances are used to process large numbers of transactions with soft real-time requirements. To summarize, large-scale multicore platforms with tens or even hundreds of cores per chip may become a reality fairly soon and applications with (soft) real-time constraints will likely be deployed on them. In this paper, we consider the issue of how to efficiently schedule soft real-time workloads on such large platforms.

In most proposed multicore platforms, different cores share on-chip caches. For example, by the end of 2007, both Intel and AMD plan to have chips with a cache shared by four cores, and the aforementioned Azul chip

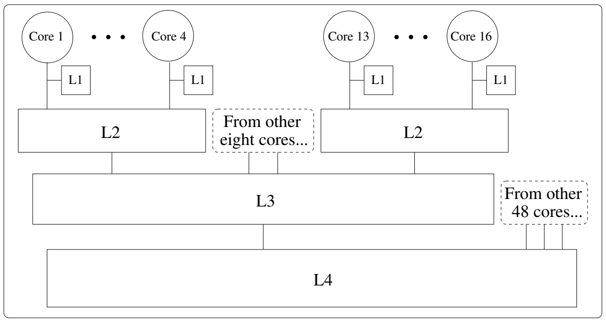

Figure 1: Large-scale multicore architecture with a four-level cache hierarchy. Three of the levels contain shared caches.

has a shared cache for each group of eight cores. To effectively exploit the available parallelism in these systems, such caches must not become performance bottlenecks. In fact, the issue of efficient cache usage on multicore platforms is one of the most important problems with which chip makers are currently grappling. This will become an even more pressing issue as the number of cores on a chip increases. For example, Intel envisions a 32-core platform where *all* cores share a cache. To alleviate issues related to cache contention and coherence, a tree-like cache hierarchy will likely exist. To reasonably constrain the focus of this paper, we henceforth take the 64-core platform shown in Fig. 1 to be the "canonical" large platform under consideration. In this platform, all cores are symmetric, single-threaded, and share an L4 cache. Note that groups of cores also share L2 and L3 caches, which reduces contention at the L4 cache.

Given such a platform, the question of whether to use partitioning or global scheduling approaches when scheduling soft real-time applications becomes more complicated. While either approach might be viable, each has serious drawbacks on this platform, and neither will likely utilize the system very well. Global scheduling algorithms are better able than partitioning approaches to utilize multiprocessor systems when system overheads are negligible. For example, on a system with M cores, the global earliest-deadline-first (EDF) algorithm can ensure bounded deadline tardiness (which is sufficient for many soft real-time applications) for any such task system if total utilization is at most M [5, 10]. On the

*Work supported by a grant from Intel Corp., by NSF grants CNS 0408996, CCF 0541056, and CNS 0615197 and by ARO grant W911NF-06-1-0425.

other hand, global algorithms are susceptible to large overheads on large platforms. These overheads are due to scheduling-related costs when scheduling a large task set on a large number of cores, high contention for the global run queue, and the cost of migrating data between two cores that share only a low-level cache (or, on some platforms, no cache at all). Partitioning approaches result in no task migrations and reduced scheduling costs; however, due to bin-packing limitations, there exist task systems with total utilization of approximately $M/2$ that no such approach can correctly schedule, even if bounded deadline tardiness is allowed. This becomes an even greater concern on large-scale platforms with relatively simple cores (a likely scenario, since using simple cores enables more cores to be placed onto a chip). This is because task utilizations on such simple cores may be high, which makes partitioning more difficult.

Contributions. Driven by the above considerations, we propose the use of a *hybrid* approach that exploits the natural groupings of cores around different levels of shared caches. In our hybrid approach, we partition the platform into *clusters* of cores that share a cache. We then statically assign tasks to clusters, and schedule tasks within each cluster using a global scheduling algorithm, namely, preemptive global EDF. In this approach, migration costs within a cluster are a function of the access time (and size) of the shared cache of that cluster. When tasks have large working sets (WSs), cluster sizes can be kept small in order to keep migration costs low. By partitioning the system into clusters instead of individual cores, we alleviate bin-packing limitations by effectively increasing bin sizes in comparison to item sizes. For example, with four-core clusters, a task can occupy at most 25% of a bin. As a result, it is much easier to partition such tasks onto clusters than onto individual cores. Note that *by tuning the cluster size, we can mitigate the weaknesses of* (*and exploit the advantages of*) *each approach*.

The "ideal" cluster size depends on both the maximum task utilization and task working set size (WSS), as well as the overall system utilization of the real-time workload. One of the main contributions of this paper is to devise rules of thumb for choosing cluster sizes. These rules were devised based upon a series of schedulability experiments that were conducted for our canonical system under a variety of different real-time workloads. We used SESC, a cycle-accurate architecture simulator that supports the MIPS instruction set [8], to obtain realistic overheads for our schedulability experiments.

We found that larger cluster sizes improve bin packings, and are preferable when task utilizations are high. When task utilizations are lower, migration and scheduling costs are usually the greater concern, particularly when WSSs are large, and therefore smaller cluster sizes are preferable. We show that for many of the workloads that we investigated, a cluster size of four is ideal for our platform, as the maximum size of a bin item is reduced to a size that makes bin packing much easier while keeping scheduling and migration costs low. There also exist scenarios where a pure partitioning or global scheduling approach is the best choice, and we do not rule out the possibility of using these approaches when it is beneficial to do so. For example, a pure partitioning approach might be beneficial when the real-time workload is very small, as the bin-packing problem would not be a concern. Alternately, when task utilizations are very high and WSSs are very low, a pure global approach might prove to be best. *Our hybrid approach simply provides greater flexibility when determining how to most efficiently schedule a real-time workload.*

One limitation of our experiments is worth noting. While we believe that many of our conclusions are of a general nature, these conclusions have been drawn based on empirical data taken from *one* simulated test platform, that shown in Fig. 1 and further elaborated upon in Sec. 3. SESC produces very detailed simulations, but is quite slow, so it was only feasible to explore one such platform in this paper. While it is difficult to predict exactly what future 64-core platforms will look like, we believe that our chosen platform is a reasonable approximation of what can be expected. In future work, we hope to evaluate other platforms, most notably those exhibiting either core or cache asymmetry.

Another difficultly in performing these experiments for *soft* real-time systems is the need to generate *average-case*, rather than worst-case, measurements. This is particularly difficult when determining system overheads, as discussed further in Sec. 3.1. Furthermore, when provisioning a system based on average-case execution costs, overruns can occur. We assume that such overruns will be temporary—if an execution cost is truly an average-case measurement, then underruns should occur sufficiently frequently to counterbalance overruns in the long run. In terms of scheduling, there are numerous ways of handling overruns. In this paper, we assume that a job that does not complete by the end of its provisioned time uses time allocated to the next job of the same task. This is the simplest way of handling overruns, as it does not require modification to the scheduling algorithm and prevents an overrunning task from having a negative impact on other tasks in the system. Unfortunately, this approach can lead to increased deadline tardiness. It also makes it difficult to consider truly non-preemptive scheduling algorithms, as it allows overrunning jobs to be preempted. Issues related to using non-preemptive scheduling algorithms assuming average-case execution costs are left as future work.

Related work. Little work has explored the impact of large-scale multicore platforms on real-time scheduling algorithms. Recent work [3] (by our research group) has compared partitioning and global approaches using real scheduling and system overheads. However, this work was performed on a conventional (non-multicore) four-processor SMP platform. Unfortunately, the platform of interest in this paper, or any approximation thereof, may not exist for at least several years, so we must resort to simulation.

What real-time workload needs 64 cores? Before continuing, we explain the need for real-time support on a platform with 64 cores. One envisioned application of multicore platforms is as *multi-purpose home appliances*, with one machine serving many of the computing needs of a home. These may include supporting, for example, HDTV video streaming, a videoconferencing session, and several "virtual" user terminals simultaneously. In such a system, applications with soft real-time requirements must run alongside other (soft real-time and non-real-time) applications. If our hybrid approach were used to implement such a system, then some tasks of the real-time workload could actually be server tasks for supporting non-real-time processing. In such a scenario, our approach would increase the capacity of the system for handling both real-time *and non-real-time* applications. It is worth noting that, in these home appliances, many of the envisioned real-time applications might be very processor-intensive (*e.g.*, streaming from an HDTV video source), and therefore might contain tasks with utilizations higher than are typically considered "normal" for real-time tasks, especially if processing cores get simpler as the number of cores on a chip increases. For this reason, we consider task sets containing such high-utilization tasks in our experiments.

Another application of these large-scale platforms is real-time transaction processing. As noted earlier, Azul [9] is a company that has developed a variety of large-scale multicore platforms explicitly for the purpose of handling transaction-oriented, Java-based workloads. Processors with 48 cores on a chip are being placed into systems with hundreds of cores in order to handle the demand created by businesses such as financial institutions. While developing more powerful hardware is certainly an important part of satisfying the demand for processing real-time transactions, it will also be important to most efficiently make use of currently-existing hardware through the use of appropriate scheduling algorithms. This will allow timing guarantees to be made while supporting the largest real-time workload possible, in turn allowing businesses to achieve the largest possible return on their hardware investment.

Organization. The rest of this paper is organized as follows. In Sec. 2, we present a brief introduction to EDF-based partitioned and global scheduling and a description of our hybrid EDF scheduling approach. In Sec. 3, we present the results of experiments performed to determine the ideal cluster size for our target platform under a variety of real-time workloads. In Sec. 4, we state several "rules of thumb" based on these results that can be employed when determining an appropriate cluster size for a real-time workload running on a large-scale multicore platform. Finally, in Sec. 5, we conclude.

2 EDF Scheduling

We focus herein on the scheduling of *periodic task systems*. Each task in such a system is invoked or *released* repeatedly; each such invocation is called a *job* of the task. A periodic task is specified by a *period*, which denotes the (exact) separation between its successive job releases, and by an *execution cost*, which denotes the maximum execution time of any of its jobs. Each job of a task has a deadline corresponding to the release time of the task's next job. Task periods are assumed to be integral with respect to the length of the system's scheduling quantum, but execution costs may be non-integral. A task's *utilization* or *weight* is given by the ratio of its execution cost and period.

In EDF scheduling algorithms, jobs are scheduled in order of increasing deadlines, with ties broken arbitrarily. In partitioned EDF (P-EDF), tasks are statically assigned to processors and those on each processor are scheduled on an EDF basis. Tasks may not migrate. In global EDF (G-EDF), jobs are allowed to be preempted and job migration is permitted with no restrictions. No variant of EDF is optimal, *i.e.*, deadline misses can occur under each EDF variant in feasible systems (*i.e.*, systems with total utilization at most the number of processors). It has been shown, however, that deadline tardiness under G-EDF is bounded in such systems, which is sufficient for many soft real-time applications [5, 10]. (In contrast to G-EDF, for global *static-priority* algorithms, scenarios exist in which tardiness is unbounded even though total system utilization is at most M [4].)

In our hybrid EDF approach, herein referred to as H-EDF, tasks are statically assigned to fixed-size clusters, much as tasks are assigned to processors in P-EDF. The G-EDF algorithm is then used to schedule the tasks on each cluster. Tasks may migrate within a cluster, but not across clusters. In other words, each cluster is treated as an independent system for scheduling purposes. Under such an approach, deadline tardiness is bounded for each cluster as long as the total utilization of the tasks assigned to each cluster is at most the number of cores per cluster.

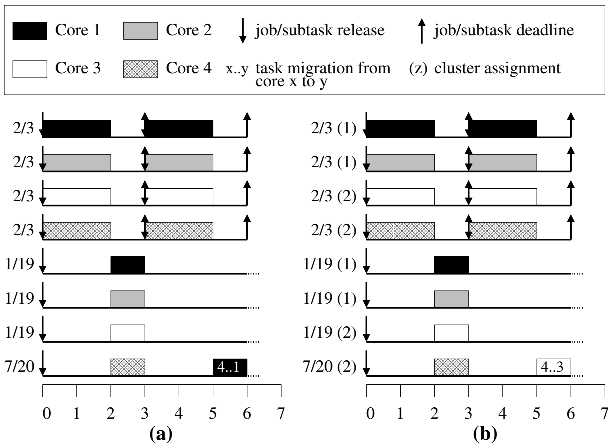

Figure 2: **(a)** G-EDF and **(b)** H-EDF schedules on a four-core system of eight tasks: four with an execution cost of 2 and period of 3, three with an execution cost of 1 and a period of 19, and one with an execution cost of 7 and a period of 20. We assume a cluster size of two under H-EDF.

To see some of the differences in these algorithms, consider Fig. 2, which depicts two schedules for a system of eight tasks, as defined in the figure's caption. These tasks are scheduled on a four-core system, with Cores 1 and 2 in one cluster and Cores 3 and 4 in another cluster. (The cores in each cluster share a cache.) There are several things worth noting here. First, these eight tasks cannot be partitioned onto four cores—each task of utilization $2/3$ needs to be placed on its own core, thus the task of utilization $7/20$ cannot be placed anywhere since $2/3 + 7/20 > 1$. As a result, this system is not schedulable under P-EDF (so we do not depict a schedule for this case). Second, under each of G-EDF and H-EDF, tasks are migrated and deadlines may be missed. Under H-EDF, tasks migrate only within a cluster, which lessens migration costs. Under G-EDF, tasks may migrate across clusters, and thus migration costs may be higher. Such higher costs in practice might result in increased deadline tardiness or the inability to schedule the task set under G-EDF. This is less likely to occur under H-EDF. Thus, H-EDF allows the task set to be scheduled (unlike P-EDF) with reduced overheads (as opposed to G-EDF).

3 Experimental Results

In this section, we report on the results of experiments conducted to compare both P-EDF and G-EDF to H-EDF with several different cluster sizes. We compared these algorithms on the basis of schedulability. The results of these experiments are presented in Sec. 3.2. In the schedulability evaluation, random task sets were generated and their schedulability under each scheme checked.

Component	**Attributes**
Processor	64 32-bit processing cores, 3 GHz clock speed
L1 instr. cache	Private, 2-way SA, 8 KB, 2 cycles/access
L1 data cache	Private, 4-way SA, 8 KB, 2 cycles/access
L2 cache	Shared (4 cores), 8-way SA, 128 KB, 8 cycles/access
L3 cache	Shared (16 cores), 8-way SA, 2 MB, 16 cycles/access
L4 cache	Shared (64 cores), 8-way SA, 32 MB, 32 cycles/access

Table 1: Platform processor and cache attributes.

In this evaluation, realistic overheads were assumed when checking schedulability. In Sec. 3.1 below, we discuss the micro-benchmarks that were used to determine these overheads.

3.1 Micro-Benchmarks

We measured four sources of overhead of relevance to each algorithm: task *preemption* and *migration* costs, and *context-switching* and *scheduling* overhead. Preemption and migration costs are dominated by the time it takes to reload data into a cold cache, and potentially invalidate data in remote caches. Context-switching overhead reflects the actual cost of switching between two tasks, and does not include any task-specific cache-related costs. Scheduling overhead reflects the cost of making one scheduling decision.

For the micro-benchmarks, time-related measurements were taken using the SESC architecture simulator. The architecture simulated is summarized in Table 1. These caches are somewhat smaller than they might be in reality—this "scaling back" was required in order to feasibly perform simulations given system memory and time constraints. Nevertheless, the cache sizes and cycle times are "realistic" in that the number of cycles required to access a level-$(k+1)$ cache is approximately twice that of a level-k cache, and the size of a level-$(k+1)$ cache is about four times the size of the *sum* of the sizes of all level-k caches that "share" this cache. For example, four L2 caches feed into an L3 cache, so the L3 cache needs to be four times the size of four L2 caches, or *sixteen* times the size of a *single* L2 cache.

The SESC simulator is very accurate, but is extraordinarily slow, particularly when simulating a 64-core platform. Because of this, it was not feasible to simulate a full 64-core platform for every scenario of interest. When cluster sizes were small, we instead simulated only a portion of the platform to get the measurements we desired. For example, with a cluster size of four, we simulated a four-core machine with a single shared L2 cache, and a "main memory" with the characteristics of an L3 cache. Since the per-task WSSs considered in this paper are not large enough to thrash the L3 cache, there was no need to simulate the L4 cache or any other part of the system for such experiments.

Preemption and migration costs. As we are interested in *average-case* costs rather than worst-case costs, given our emphasis on soft real-time applications, measuring preemption and migration costs was not as straightforward as might be expected. (In the worst-case scenario, all reads/writes are from/to memory, so knowledge of the cache hierarchy is not needed when measuring preemption or migration costs.) Under each algorithm, aligned quanta represent the worst-case scenario in terms of bus contention. Such an alignment will occur at least once per hyperperiod. We measured preemption and migration costs by focusing on one quantum of execution on each core, with all quanta beginning at some time $X + k$, where X is the same for all processors (synchronized using a barrier), and k is a small amount of random wait time (different for each core). This wait time was between 0 and 1,000 iterations of an empty loop, equivalent to no more than approximately one microsecond of wait time. This wait time, however, was large enough to add a necessary random element to the simulation runs, which would otherwise generate identical results over multiple runs. This wait time also allowed all cores to be busy while placing less bus contention pressure on the system than would be expected in the worst case.

We used SESC to measure the cost of a preemption (under P-EDF) or a migration (under G-EDF and H-EDF), while varying both cluster size and the amount of time that a task was preempted, in quanta. We assumed that preemption and migration costs did not increase substantially for preemption periods of five quanta or more, since virtually all of the cache lines associated with the WS of a task will have been evicted in most or all cache levels after such a time period. Therefore, we did not measure these costs for preemption periods greater than five quanta. (Later, when we use these measurements, we assume that the preemption or migration cost for a task that is preempted for greater than five quanta is identical to the cost assuming that task is preempted for *exactly* five quanta.)

The actual cost of a preemption or migration was measured as follows. First, the WS of the task was read, bringing data from the WS into the cache hierarchy. Second, the task was preempted, and other data was brought into the cache hierarchy at the same rate for some number of quanta (indicating the preemption period of the task). Finally, the time to sequentially write the WS of the task from the same processor (simulating a preemption under P-EDF) or from another processor *in the same cluster* (simulating a migration under G-EDF or H-EDF) was measured. For smaller preemption periods, this time should be smaller, since more data from the WS should be present in the cache hierarchy, as fewer cache lines are evicted, thus resulting in improved performance. We then subtracted from this time the time to write the same WS assuming *no* preemption occurred, *i.e.*, assuming the maximum level of cache reuse. The resulting time represents the additional overhead resulting from the task preemption or migration—in other words, the preemption/migration cost. Note that, in the case of a migration, our measurement method may force cache invalidations at several caches in the hierarchy, resulting in an additional cost. This cost can be significant, especially during a write, due to the synchronous nature of cache coherency protocols, and therefore we cannot simply assume that preemption and migration costs are equivalent. In all cases, we assumed that the data was migrated through the lowest level of cache shared by all cores in the cluster. For example, with a cluster size of four, the lowest level of cache shared exclusively by the cores within a cluster is an L2. As a result, when using smaller cluster sizes, it was much more likely that any cache reuse would result in improved performance, since cache access times were smaller and the shared cache of the cluster was "closer" to its cores.

One reason that we chose to measure the cost of writing the WS rather than reading it was to ensure that we captured the full impact of a task migration, as cache invalidations are the primary source of additional overhead when a task incurs a migration versus a preemption. Such invalidations should only be generated by a write following a read—a read following a read would not require a cache invalidation in most cases.

Task WSSs were varied over $\{4\text{K}, 32\text{K}, 64\text{K}\}$ bytes in the micro-benchmark results. Larger WSSs were not chosen for several reasons. First, simulating larger task WSSs took an extraordinarily long time in SESC. Second, we define WSS with respect to a *single quantum of computation*, and it is not possible to write too much more than 64K bytes within a reasonably-sized (*e.g.*, 1-ms) quantum. Finally, these WSSs are intended to be a measure of *cache reuse*, and not the total memory usage of a task during a quantum—if after a preemption or migration a task accesses memory that was never in any cache, then this access is part of the task's execution cost and does not contribute to preemption or migration costs. Many applications have high locality, so their WSS is small (in terms of reuse), even if the amount of data they access in a quantum is large. Overall, the overheads that we measured are reasonable and allowed us to get meaningful numbers for our purposes so that a valid comparison of EDF approaches and cluster sizes could be made on our platform.

Generating average-case costs. Using the preemption and migration cost data obtained using SESC, for each combination of cluster size, WSS, and preemption period, we then generated *average-case* preemption and migration costs. For each combination of cluster size, maximum task utilization, and system utilization shown in Table 2, we simulated the execution of 100 randomly-generated task sets until their hyperperiods. For each task set, a frequency distribution was generated indicating the lengths of the preemption periods tasks experienced during execution. These frequency distributions were then combined into one distribution, which was used to calculate a *weighted* average preemption or migration cost for that combination. As a simple example, assume that after simulating all task sets, we get a frequency distribution indicating that 1,000 preemptions were two quanta long, and 500 were ten quanta long. If the preemption/migration cost for a preemption period of two quanta is 50 μs, and for a period of five or more quanta is 100 μs, then the resultant average preemption/migration cost would be $(50 * 1000 + 100 * 500)/(1000 + 500) \approx 66.67\mu s$.

Finally, it is worth noting that our approach to generating average-case costs is only one way that such costs could be generated. Determining the "average case" for arbitrary task sets is difficult due to the subjective nature of determining what is average for a particular class of applications. However, we believe that our approach of combining a frequency distribution of preemption periods and measured preemption and migration costs from a simulated platform is general enough to provide a reasonable estimate of average-case costs under a variety of workloads and scheduling algorithms.

Results from these experiments, in the form of average preemption and migration costs for all combinations, appear in Table 2. The notation H-EDF-Cx in the table indicates the H-EDF scheduling algorithm with a cluster size of x. There are several trends to note here. First, average preemption and migration costs scale approximately linearly with WSS, as expected. Second, average preemption and migration costs increase significantly as cluster sizes increase, since larger clusters share a larger, slower cache than smaller clusters. The effect is particularly dramatic as cluster size increases from four to sixteen and again from sixteen to 64. Third, note that system utilization and maximum task utilization have little to no impact on preemption and migration costs. Since these experiments measured preemption and migration costs over a wide range of system and task utilizations, we averaged the costs across each column in the table, and used these values in the schedulability experiments in Sec. 3.2.

Context-switching overhead. We measured context-switching overhead by reading the time it took a SESC thread to switch from one task to another. Each core ran a single thread that was pinned to that core, and these threads were responsible for executing tasks on that core. Due to limitations of the SESC thread library, "real" context switches could not be performed.[1] Switching from one task to another was relatively straightforward, as it only involved the reassignment of several variables and pointers. As a result, we never observed context-switching overheads greater than one microsecond, and the overhead was often negligible. These overheads are similar to those measured on a real platform in [3], which contains only slightly slower processors than the cores in the platform simulated in this paper. We believe this serves to validate our simulated measurements. In the schedulability experiments that follow, we conservatively assume that all context switches require exactly one microsecond.

Scheduling overhead. Due to SESC-related time constraints, we generated reasonable values for scheduling overheads analytically, using some empirical data, rather than through an entirely experimental procedure. We are interested in *average-case* scheduling overheads representing the average cost of a single scheduling decision. Consider first the cost of a scheduling decision made between quantum boundaries. Assuming integral periods, new jobs will only be released by (synchronous) periodic tasks at quantum boundaries. Therefore, only scheduling decisions resulting from job completions will be made between quantum boundaries. As these job completions are typically distributed evenly over time, it is unlikely that these scheduling decisions will result in contention for run queues or other scheduler-related structures, which would result in additional overhead. Moreover, the scheduling decisions themselves are of negligible cost individually, as they only involve a single dequeue from the run queue. Thus, we considered the cost of scheduling decisions made between quantum boundaries to be negligible (at most 1-2 μs).

On the other hand, scheduling decisions made *at* quantum boundaries incur overheads that are *not* negligible, because contention may occur, thus forcing cores to wait to acquire a queue lock to access the task run queue. The overhead (in ns) associated with these scheduling decisions can be calculated as follows. Let γ represent the kernel overhead of entering a queue lock, let c represent the average number of jobs that are dequeued from the run queue and scheduled at a quantum boundary *within a single cluster* (or a single core for P-EDF or all cores for G-EDF), and let α represent the cost of scheduling one job. Then, at a quantum boundary, c cores will dequeue and schedule one job, while the remaining cores find an empty run queue. We assume that the cost of checking an empty run queue should only involve checking a pointer

[1] If "real" context switches were performed, and tasks did not share an address space, then some additional overhead may be incurred to invalidate and repopulate the TLB, and to load into cache the page-table entries of the task that is being switched to.

Sys. Util.	Max. Task Util.	P-EDF	H-EDF-C4	H-EDF-C16	G-EDF
25%	1/10	0.00	0.07	3.63	6.88
50%	1/10	0.01	0.08	3.54	6.85
75%	1/10	0.03	0.09	3.74	6.64
100%	1/10	0.04	0.13	3.99	6.60
25%	1/5	0.00	0.09	3.89	6.78
50%	1/5	0.00	0.07	3.51	6.89
75%	1/5	0.01	0.07	3.56	6.72
100%	1/5	0.03	0.10	3.79	6.66
25%	1/3	0.00	N/A	3.85	6.74
50%	1/3	0.00	0.08	3.46	6.91
75%	1/3	0.01	0.08	3.52	6.84
100%	1/3	0.03	0.09	3.62	6.84
25%	1/2	0.00	N/A	3.65	6.70
50%	1/2	0.00	0.08	3.66	6.87
75%	1/2	0.01	0.07	3.46	6.88
100%	1/2	0.02	0.09	3.56	6.89
25%	1	0.00	N/A	3.80	6.86
50%	1	0.00	0.08	3.58	6.83
75%	1	0.01	0.08	3.71	6.82
100%	1	0.02	0.09	3.59	6.78
Averages		**0.01**	**0.08**	**3.66**	**6.80**

(a)

Sys. Util.	Max. Task Util.	P-EDF	H-EDF-C4	H-EDF-C16	G-EDF
25%	1/10	17.98	21.27	37.38	72.06
50%	1/10	19.24	22.27	36.09	71.95
75%	1/10	19.67	23.35	38.09	71.53
100%	1/10	19.81	24.21	40.59	71.86
25%	1/5	17.55	21.50	39.96	72.18
50%	1/5	18.95	21.87	35.94	72.08
75%	1/5	19.71	22.81	36.26	71.71
100%	1/5	19.94	23.67	38.56	71.54
25%	1/3	18.06	N/A	39.43	72.23
50%	1/3	18.77	21.58	35.36	72.01
75%	1/3	19.19	22.50	35.84	71.91
100%	1/3	19.67	23.01	36.72	71.97
25%	1/2	17.62	N/A	37.44	72.21
50%	1/2	18.19	21.86	37.46	72.01
75%	1/2	19.16	21.94	35.56	71.90
100%	1/2	19.10	22.44	36.64	71.60
25%	1	17.12	N/A	38.70	71.65
50%	1	18.03	21.77	36.86	71.77
75%	1	17.73	22.95	37.83	71.85
100%	1	18.30	22.60	36.82	71.48
Averages		**18.69**	**22.45**	**37.38**	**71.88**

(b)

Sys. Util.	Max. Task Util.	P-EDF	H-EDF-C4	H-EDF-C16	G-EDF
25%	1/10	37.71	39.30	93.18	128.85
50%	1/10	34.18	38.42	92.86	130.81
75%	1/10	30.18	36.98	102.59	133.43
100%	1/10	27.80	34.94	110.36	132.21
25%	1/5	38.12	38.52	103.32	129.10
50%	1/5	35.51	38.65	90.44	130.01
75%	1/5	32.60	38.22	96.45	133.03
100%	1/5	29.12	36.40	103.86	133.23
25%	1/3	36.85	N/A	101.46	129.32
50%	1/3	35.75	39.02	88.07	129.83
75%	1/3	34.32	38.14	92.19	131.96
100%	1/3	30.99	37.07	96.12	131.05
25%	1/2	38.06	N/A	94.19	129.79
50%	1/2	36.60	38.65	95.39	129.50
75%	1/2	34.14	38.88	88.92	130.79
100%	1/2	32.43	38.00	92.77	130.25
25%	1	38.53	N/A	103.97	129.88
50%	1	36.85	38.68	93.82	130.52
75%	1	36.23	37.71	99.71	131.15
100%	1	33.91	37.73	95.07	132.00
Averages		**34.49**	**37.96**	**96.74**	**130.84**

(c)

Table 2: Average preemption/migration costs (in μs) for per-task WSSs of **(a)** 4KB; **(b)** 32KB; and **(c)** 64KB. An "N/A" entry indicates that after simulating 100 task sets given the specified system utilization, maximum task utilization, and cluster size, no preemptions occurred—task utilizations were high enough, and system utilization low enough, that in most cases, there were fewer tasks than cores. Thus, there was no way to generate average preemption and migration costs.

or flag, and therefore its cost is negligible. A core waiting for the queue lock will, on average, have to wait for approximately half of the cores in the cluster to acquire the queue lock before acquiring it itself. As a result, it will incur the cost of the scheduling decisions made by half of the cores. Thus, the average scheduling overhead at quantum boundaries is $\gamma + c\alpha/2$.

When using this formula to calculate scheduling overheads, we set γ to $750ns$ and α to $1250 + 125 \cdot log_2(num\_tasks/num\_clusters)\ ns$. These values are based on overheads empirically measured on a real platform in [4] (the same platform used in [3]). The processors of this platform have similar performance characteristics to the cores in our platform, and therefore should be valid for our purposes. The expression for α was derived as follows. First, a constant base cost is incurred when removing a job from the head of the run queue and beginning its execution, which in this case takes $1250ns$. An additional cost is incurred when initially adding a job to the run queue. This cost is logarithmically related to the average number of tasks assigned to a given cluster, or $num\_tasks/num\_clusters$, assuming an efficient binomial help implementation of the needed priority queues. The base cost of $1250ns$ and the multiplier of 125 were both again determined by empirical measurements performed as part of the work in [4].

Finally, we determined the value of c, or the average number of jobs that are dequeued and scheduled at a quantum boundary. Let j be the average number of jobs released at a quantum boundary on a single cluster. Each released job may preempt another job, which will then be placed in the run queue, to be dequeued and rescheduled at some later time. As a result, every released job ultimately results in either one or two dequeue and scheduling operations. Thus, $j \leq c \leq 2 \cdot j$. We argue that c lies considerably closer to j than $2 \cdot j$, as the majority of preempted jobs will be rescheduled *between* quantum boundaries. (Recall our earlier claim that scheduling overheads are negligible between quantum boundaries.) Therefore, we assumed the common case, *i.e.*, $c = j$.

When generating random task sets as part of the schedulability experiments in Sec. 3.2, we calculated j as part of determining the scheduling overhead. The value of j was calculated by dividing the total number of jobs released by all tasks over a large time interval (*e.g.*, 1,000,000 quanta) by the number of quanta in that interval. This provides us with an average number of jobs released at each quantum boundary. We then divided this value by the number of clusters to determine the average number of jobs released on each cluster at each quantum boundary. We initially sought to use task set hyperperiods as our time interval; however, some of our task sets were very large, resulting in a very large hyperperiod that was prohibitive to calculate and use due to arithmetic overflow issues. We instead chose an interval of one million quanta, which should produce very similar results.

The range of overheads generated (for randomly-generated task sets) with this method are shown in Table 3 for each cluster size. As expected, large cluster sizes result in additional scheduling overhead, due both to increased contention for the queue lock protecting each run queue and an increase in the number of tasks that are assigned to each cluster.

3.2 Comparison of Schedulability

To assess differences in schedulability, we determined the schedulability of 100 randomly-generated task sets for different {cluster size, per-task utilization, per-task WSS} combinations, using the overheads computed in Sec. 3.1. Cluster sizes were one (P-EDF), four, sixteen, or 64 (G-EDF). In each experiment, task utilizations were uniformly distributed over the ranges $[0.001, 0.05]$, $[0.001, 0.1]$ $[0.001, 0.6]$, $[0.001, 0.9]$, $[0.4, 0.6]$, $[0.51, 0.6]$ or $[0.6, 0.9]$; or bimodally distributed uniformly over $[0.001, 0.05)$ with probability $9/10$, and over $[0.95, 0.999]$ with probability $1/10$. Task periods were uniformly distributed over $[10, 100]$ (all units are in ms). Task execution costs were calculated from periods and utilizations. This approach to task set generation is similar to that proposed by Baker [2]. However, our utilization distributions are somewhat different in order to investigate schedulability over a broader range of distributions, including those where the performance differences between P-EDF and G-EDF would be particularly dramatic.

	P-EDF	H-EDF-**C4**	H-EDF-**C16**	G-EDF
Min.	1.38	1.38	1.50	1.63
Avg.	1.47	1.83	3.62	11.82
Max.	1.63	3.75	12.00	47.00

Table 3: The range of calculated scheduling overheads for each cluster size, in μs.

Finally, per-task WSSs were 4K, 32K, or 64K. All task sets were generated to fully utilize the system, assuming that system overheads were negligible. When overheads are included, these task sets will require more than 64 cores.

The definition of a *correct schedule* for soft real-time systems requires that deadline tardiness be *bounded* (regardless of how high the bound may be), rather than that all deadlines be met. We assessed differences in schedulability for each (100-task) experiment by computing the *average minimum required number of processors* (RNP) for producing a correct schedule.

When determining schedulability under each algorithm, we first inflated the execution cost of each task to account for the overheads discussed in Sec. 3.1 using standard techniques. These techniques are described at length in [4]. (Note that, even when RNP exceeds 64, we still only consider overheads as computed on our 64-core platform. This is perhaps one limitation of our experimental methodology.) We determined whether a task set could be scheduled on our platform as follows. For G-EDF, since we are only concerned with soft real-time schedulability in this paper, only a check that total utilization is at most M was required. For P-EDF, we determined whether each task set could be partitioned using the *first-fit decreasing* heuristic. (While a closed-form test is available for P-EDF [7], our approach is less pessimistic.) For H-EDF, we determined whether each task set could be partitioned *onto clusters*, again using the first-fit decreasing heuristic. In other words, this was

Task Util. Range	P-EDF	H-EDF-C4	H-EDF-C16	G-EDF
Bimodal Dist.	65.00	65.00	65.00	65.68
Unif. $[0.001, 0.05]$	65.00	65.00	66.00	71.00
Unif. $[0.001, 0.1]$	65.00	65.00	65.00	68.00
Unif. $[0.001, 0.6]$	65.00	65.00	65.00	65.00
Unif. $[0.001, 0.9]$	66.80	65.00	65.00	65.00
Unif. $[0.4, 0.6]$	68.50	66.88	65.17	65.00
Unif. $[0.51, 0.6]$	114.91	68.14	65.17	65.00
Unif. $[0.6, 0.9]$	85.36	68.47	65.73	65.00

(a)

Task Util. Range	P-EDF	H-EDF-C4	H-EDF-C16	G-EDF
Bimodal Dist.	65.00	65.00	65.05	66.40
Unif. $[0.001, 0.05]$	66.00	66.00	68.00	75.84
Unif. $[0.001, 0.1]$	65.00	66.00	67.00	70.00
Unif. $[0.001, 0.6]$	65.00	65.00	65.00	65.00
Unif. $[0.001, 0.9]$	66.94	65.00	65.00	65.00
Unif. $[0.4, 0.6]$	68.98	66.93	65.27	65.00
Unif. $[0.51, 0.6]$	114.99	68.21	65.19	65.00
Unif. $[0.6, 0.9]$	85.29	68.56	65.82	65.00

(b)

Task Util. Range	P-EDF	H-EDF-C4	H-EDF-C16	G-EDF
Bimodal Dist.	65.00	65.00	66.04	67.10
Unif. $[0.001, 0.05]$	67.00	67.05	72.06	79.93
Unif. $[0.001, 0.1]$	66.00	66.00	69.00	72.02
Unif. $[0.001, 0.6]$	65.00	65.00	65.00	65.98
Unif. $[0.001, 0.9]$	66.91	65.00	65.00	65.00
Unif. $[0.4, 0.6]$	69.47	66.91	65.59	65.00
Unif. $[0.51, 0.6]$	114.94	68.23	65.43	65.00
Unif. $[0.6, 0.9]$	85.37	68.55	65.88	65.00

(c)

Table 4: RNP for per-task WSSs of **(a)** 4KB; **(b)** 32KB; and **(c)** 64KB.

similar to the approach used for P-EDF, but with fewer larger bins. As the total utilization within each cluster was at most the number of cores in the cluster, the soft real-time schedulability of the workload on each cluster was guaranteed, since G-EDF is used within each cluster to schedule the tasks on that cluster.

Table 4 shows RNP results for soft real-time task sets for all combinations of the parameters listed above. There are several things to note here. First, when maximum task utilizations are low, P-EDF performs the best and G-EDF performs the worst, due both to a large number of task migrations and high scheduling costs related to a large number of tasks in each task set. As WSS increases, the cost of each migration increases, resulting in even higher RNP values for G-EDF when task utilizations are low. When maximum task utilizations are high, the situation is reversed—a decreasing number of task migrations and tasks per task set combined with the limitations of bin-packing when task utilizations are high result in lower RNP values for G-EDF than P-EDF. This result even holds for the 64K WSS case (inset (c)). Interestingly, note that the $[0.51, 0.6]$ range is even worse for P-EDF than the $[0.6, 0.9]$ range. This makes sense, since these task sets contain, on average, the largest number of tasks that require their own processor under P-EDF. Third, note that H-EDF with a cluster size of four appears to approach the performance of both P-EDF at low task utilizations and G-EDF at high task utilizations. This is because a cluster size of four results in lower scheduling costs—due to a relatively small number of tasks to consider for each scheduling decision—and lower migration costs—due to the presence of a high-level, fast shared cache. Additionally, a cluster size of four results in large enough bins to effectively avoid the limitations of bin-packing, even for task utilization ranges that are highly problematic under P-EDF. In the cases where task utilizations are exclusively low or high, the performance benefits are dramatic, and *in all cases,* H-EDF *with a cluster size of four performs similarly to the best-performing algorithm.* Note that H-EDF with a cluster size of sixteen does not result in the same performance—its cluster size is not small enough to keep migration and scheduling costs low. However, it performs slightly better than H-EDF with four-core clusters when task utilizations are high.

4 Cluster Size Guidelines

In this section, we state several guidelines when determining the cluster size for a system, given various characteristics about the real-time workload, based on the experiments performed in this paper.

Guideline 1. Unless task utilizations are very high, H-EDF is the preferable scheme to use over G-EDF. Even when cluster sizes are small, increasing bins to include several cores rather than just one is usually enough to alleviate the limitations of bin packing. When task utilizations are high, the performance of H-EDF is slightly worse, but generally acceptable except when *system* utilization is very high. When task utilizations are low, H-EDF with a cluster size of four performs much better than G-EDF.

Guideline 2. Only choose P-EDF if task utilizations are very low and system utilization is very high. Otherwise, H-EDF with a small cluster size (four in our experiments) performs almost identically when task utilizations are low, and much better than P-EDF when task utilizations are high.

Guideline 3. If tasks may change their utilizations at run time, then H-EDF with a small cluster size is the safest choice, since it has the most consistent performance over all types of workloads, including those with low-utilization tasks. It should perform comparably to

P-EDF at lower task utilizations and G-EDF at higher task utilizations.

Guideline 4. In most cases, it is beneficial to choose a cluster size equal to the number of cores that share the first level of shared cache (usually an L2 cache). Doing so keeps migration and scheduling costs low, and should greatly alleviate bin-packing issues. Even if the first level of shared cache is shared by only two cores, doubling the bin size by using a cluster size of two would improve bin-packings substantially, by ensuring that no task can occupy more than half of a single bin. Note in Table 4 that, in some cases, H-EDF with a cluster size of four had RNP values that were as much as 40% lower than P-EDF, and as much as 16% lower than G-EDF.

5 Concluding Remarks

In this paper, we have proposed a hybrid approach called H-EDF for scheduling real-time tasks on large-scale multicore platforms with hierarchical shared caches. This approach exploits the natural groupings of cores around different levels of shared caches, and avoids the fundamental limitations of both P-EDF and G-EDF on large-scale multicore platforms with many simple cores. We also determined the "ideal" cluster size for our platform, given various characteristics of the real-time workload such as task utilizations or WSSs. We regard the main contribution of this paper to be the presentation of an algorithm that both avoids the pitfalls of the less-flexible P-EDF and G-EDF algorithms for large-scale multicore platforms and can be adjusted to make the most efficient use of a wide variety of such platforms running many different kinds of real-time workloads.

In producing these results, the development of a reasonable methodology for assessing average-case costs was *by far* the most demanding and time-consuming task that we faced. We acknowledge that alternate methodologies that deal differently with some of the issues involved could produce somewhat different results. Nevertheless, we believe that the costs derived in this paper are reasonable. This data might possibly be useful to other researchers who are interested in the impact such overheads have on schedulability.

There are numerous directions for future work. First, we would like to extend our investigation to other scheduling algorithms, particularly "semi-non-preemptive" and static-priority algorithms. While we had originally intended our study to include the non-preemptive global EDF scheduling algorithm, the notion of supporting tasks with average-case execution costs becomes more difficult with non-preemptive scheduling approaches, as discussed in Sec. 1. Instead, we need to develop a "semi-non-preemptive" restricted migration algorithm that can handle overruns while retaining some of the benefits of not allowing job preemption in most cases. Second, we want to re-assess the overheads considered in this paper on other architectures, particularly platforms with core or cache asymmetry. Third, we would like to experiment with a greater variety of workloads than those we have considered to date, and experiment with different cache coherency protocols, perhaps including those not supported by SESC. Fourth, we would like explore the synchronization issues that occur when tasks are not independent. Finally, in this paper, we have assumed that bounded deadline tardiness is sufficient for supporting the soft real-time applications of concern to us. We would also like to investigate notions of soft real-time computing that take other factors into account, such as the percentage of deadlines missed, or a job's "utility" after its deadline.

References

[1] S. Bisson. Azul announces 192 core Java appliance. http://www.itpro.co.uk/servers/news/99765/azul-announces-192-core-java-appliance.html, 2006.

[2] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. Technical Report TR-051101, Department of Computer Science, Florida State University, 2005.

[3] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 2006.

[4] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006.

[5] U. Devi and J. Anderson. Tardiness bounds for global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-time Systems Symposium*, pages 330–341, 2005.

[6] C. Farivar. Intel Developers Forum roundup: four cores now, 80 cores later. http://www.engadget.com/2006/09/26/intel-developers-forum-roundup-four-cores-now-80-cores-later/, 2006.

[7] J. Lopez, M. Garcia, J. Diaz, and D. Garcia. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro Conference on Real-time Systems*, pages 25–33, 2000.

[8] J. Renau. SESC website. http://sesc.sourceforge.net.

[9] Azul Systems. Azul compute appliances. http://www.azulsystems.com/products/compute_appliance.htm, 2006.

[10] P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by EDF on multiprocessors. In *Proceedings of the 26th IEEE Real-time Systems Symposium*, pages 311–320, 2005.