MARACAS: A Real-Time Multicore VCPU Scheduling Framework

Ying Ye, Richard West, Jingyi Zhang and Zhuoqun Cheng

Computer Science Department
Boston University
Email: {yingy,richwest,jyzhangr,czq}@cs.bu.edu

***Abstract*—This paper describes a multicore scheduling and load-balancing framework called MARACAS, to address shared cache and memory bus contention. It builds upon prior work centered around the concept of virtual CPU (VCPU) scheduling. Threads are associated with VCPUs that have periodically replenished time budgets. VCPUs are guaranteed to receive their periodic budgets even if they are migrated between cores. A load balancing algorithm ensures VCPUs are mapped to cores to fairly distribute surplus CPU cycles, after ensuring VCPU timing guarantees. MARACAS uses surplus cycles to throttle the execution of threads running on specific cores when memory contention exceeds a certain threshold. This enables threads on other cores to make better progress without interference from co-runners. Our scheduling framework features a novel memory-aware scheduling approach that uses performance counters to derive an average memory request latency. We show that latency-based memory throttling is more effective than rate-based memory access control in reducing bus contention. MARACAS also supports cache-aware scheduling and migration using page recoloring to improve performance isolation amongst VCPUs. Experiments show how MARACAS reduces multicore resource contention, leading to improved task progress.**

I. INTRODUCTION

There is an increasing prevalence of multicore processors in embedded and real-time systems. These processors offer power and performance benefits over single-core alternatives running at higher clock frequencies. However, complex on-chip cache hierarchies, including shared last-level caches, and memory buses that are common to all cores, pose challenges for tasks with real-time requirements. A task running on one core may experience harmful contention for cache lines that are shared with tasks running on other cores. Consequently, a task that should seemingly run in isolation of tasks on other cores experiences timing unpredictability due to unforeseen cache line evictions, misses, and reloads. Similarly, cache-line fills with instructions and data require accesses to a shared memory bus. This may lead to one or more co-running tasks being forced to stall while a memory bus transaction is performed for another task.

This paper describes a multicore scheduling and load-balancing framework, called *MARACAS* [1], applied to our Quest real-time operating system. It extends our prior work on uniprocessor real-time virtual CPU (VCPU) scheduling [1], to deal with micro-architectural challenges associated with multicore processors. Quest associates one or more task threads with VCPUs, which in turn are scheduled on physical CPUs. This hierarchical approach is similar to that found in virtual machine systems, decomposing the scheduling problem into simpler layers. Rather than tracking time for every individual thread in a system, it is only necessary to track the time usage of VCPUs. Real-time threads can be assigned dedicated VCPUs, while threads that are not time critical can be scheduled with other threads on the same VCPU. Each VCPU is guaranteed a budgeted amount of CPU time in a specific period, ensuring bandwidth preservation. By managing CPU time at the granularity of VCPUs rather than threads, we reduce the overhead of reprogramming the hardware timers to track available budget usage.

In Quest, each core is associated with a separate VCPU scheduling queue, but VCPUs are *migratory* between cores. The scheduling approach ensures that every VCPU created in the system is guaranteed its *foreground* budget. Once a VCPU has exhausted its budget, it cannot execute in foreground mode again until it is replenished at least part of its budget some period of time in the future. If every VCPU on a processor core has exhausted its budget then the core switches to *background* mode. Essentially, background mode is a state in which there are surplus CPU cycles on the corresponding core.

In this paper, we show how the MARACAS scheduling framework uses background cycles to improve system performance (e.g., to maximize the total instruction execution count) and to balance resource usage across a set of cores. This means the system guarantees timing constraints on VCPUs while attempting to maximize the progress of a set of tasks. We see this as being beneficial to applications that improve the resolution, or quality, of their results when granted extra computation time. For example, with "anytime" computing [2], or imprecise computations [3], a task is divided into *mandatory* and *optional* parts: execution of an optional part proceeds to improve quality if there are sufficient resources. This could apply to an MPEG-encoded video stream where it is mandatory to process I-frames, but optional to process B- and P-frames that improve frame rate. Similarly, an obstacle avoidance system might use a mandatory time allocation to identify potential collisions with objects whose locations are estimated, while extra compute cycles improve the accuracy of various obstacle positions.

MARACAS addresses shared cache and memory bus contention, while ensuring task timing requirements. Page col-

[1] Memory-Aware, Real-time-Aware, Cache-Aware Scheduling.

oring techniques ensure that address spaces associated with specific VCPUs map to cache lines that do not conflict with other address spaces. MARACAS uses page color information as part of its cache-aware load balancing strategy, to maximize the instructions executed per cycle on the corresponding VCPU. A significant contribution of this work is a novel memory throttling approach for each core operating in background mode. As part of MARACAS' memory-aware scheduling, each core uses hardware performance counters to monitor the *average memory request latency*. If this exceeds a specified latency threshold, then the core prevents usage of background cycles until the memory access rate across all cores decreases below a rate threshold. We show by a series of experiments how these aspects of cache- and memory-aware scheduling improve system-wide performance for a series of task sets, where all VCPUs are guaranteed their timing requirements. Our latency-based memory throttling approach is shown to be more effective than rate-based memory access control at reducing bus contention.

The next section provides brief background information about the Quest VCPU scheduling framework. This is followed by several sections that describe the memory- and cache-aware scheduling features that are new to MARACAS, including the algorithms for VCPU load balancing and background-mode resource management. Experiments are described in Section VI, followed by Related Work in Section VII. Finally, Conclusions and Future Work are discussed in Section VIII.

II. Quest Operating System

A. VCPU Scheduling

The Quest real-time system implements a novel virtual CPU (VCPU) scheduling framework [1]. Rather than scheduling threads directly on physical CPUs or cores (PCPUs), the scheduling problem is decomposed into a simpler two-level hierarchy (Figure 1). One or more threads are assigned and scheduled on VCPUs, which are then scheduled on PCPUs. This way, groups of threads that are non-time-critical or which are part of an equivalent class can share a single VCPU, while specific real-time tasks may be assigned separate VCPUs.

VCPUs are resource containers [4] for threads that are assigned to them. They account for budget usage in specific windows of real-time. By default, each VCPU is specified a processor capacity reserve [5] consisting of a budget capacity, C, and period, T. A VCPU is required to receive at least its budget every period when it is runnable. The Quest scheduling framework distinguishes between VCPUs for handling task execution and system events, such as interrupts. However, for this paper, we assume that every VCPU is implemented as a Sporadic Server [6], [7], and each VCPU is assigned a single thread.

Each core is associated with a separate scheduling queue. All VCPUs assigned to the same core are scheduled using Rate-Monotonic Scheduling (RMS) [8]. The RMS utilization bound is then applied on a per-core basis when assigning VCPUs to cores. Schedulability tests are performed when new VCPUs are created and when they are migrated between cores.

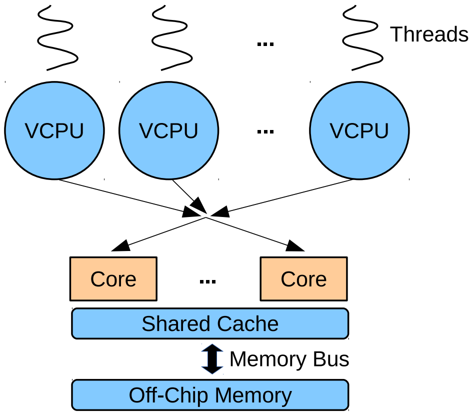

Fig. 1. VCPU Framework

B. Performance Monitoring

Quest features a performance monitoring sub-system, which uses hardware counters to track system events. Cycle-accurate timestamp counters track the CPU time usage of each and every VCPU. Additional counters are configurable via an API to monitor events such as cache references, misses, hits, and instructions retired. We use these event counts for cache occupancy estimates [9], [10] and cache-aware scheduling. Cache occupancy estimates make it possible to determine the correct sizes of cache partitions to avoid contention.

C. Cache Partitioning

On multicore platforms, shared cache contention is a significant problem for real-time systems. Contention is eliminated by partitioning a cache into separate regions for each task. Software approaches to cache partitioning include page coloring [11]. Pages of different colors are mapped by hardware to different cache lines [2]. Quest has the option of being configured and built with support for a color-aware memory allocator [12], to assign pages of memory to a task. The allocator maintains separate lists of free pages for each color. Each core is associated with a different subset of these lists and, hence, page colors. This ensures that tasks running on separate cores do not experience shared cache conflicts.

III. Background Scheduling

Each VCPU with available budget at the current time operates in *foreground* mode. When a VCPU depletes its budget it enters *background* mode, where it will only be scheduled if there are no other runnable VCPUs in foreground mode on the same core. A core is said to be in background mode when all VCPUs assigned to it are in background mode. At this point, the core invokes its background scheduling policy. MARACAS implements a background scheduling algorithm that attempts to fairly distribute surplus CPU time amongst VCPUs. Every task and, hence, VCPU [3] is tracked for the amount of background CPU time (BGT) it has used so far. When a core enters background mode, the local scheduler

[2] Or at least different sets of cache lines in set-associate caches.

[3] Unless otherwise stated, we use "task" and "VCPU" interchangeably.

picks a task with the smallest BGT and keeps it running until the core switches back to foreground mode. The mode switch occurs when a VCPU is replenished with available budget.

An alternative background scheduling approach in MARACAS is to keep the same task running on a core when it switches from foreground to background mode. If that task happens to block during background mode, the system schedules the task that is expected to run first when the core switches back into foreground mode. This method attempts to reduce context switches, but experimental results suggest there is negligible performance benefit.

A further background scheduling option in MARACAS attempts to reduce cache and memory bus contention. A scheduler running on a core in background mode gives precedence to tasks that are less memory intensive. This approach guarantees budgeted foreground time for a set of tasks, while trying to use surplus CPU time to minimize resource contention. Other approaches have attempted to co-schedule tasks to avoid cache and memory bus contention, at the cost of meeting timing requirements [13], [10]. The use of separate foreground and background modes enables VCPU timing constraints to be met, while allowing for objectives such as fairness and performance to be addressed.

MARACAS' use of VCPUs is similar to *soft reservations* in the resource kernel [14]. When a soft reservation is depleted, it can be scheduled for execution along with unreserved threads and other depleted reservations. Similarly, when a foreground VCPU in MARACAS depletes its budget, it is possible to schedule that VCPU along with others on the same core when there are no more eligible to execute in foreground mode. However, MARACAS implements novel background scheduling policies to explicitly address multicore resource contention by carefully regulating which cores are eligible to use their background time. Some cores are throttled from using their background time to allow others to make progress without contention for the memory bus. Others have developed slack stealing algorithms [15] to enable soft real-time tasks to acquire resources while ensuring hard real-time task guarantees. Our use of background time is to improve task progress beyond the base-level provided by foreground timing guarantees.

IV. Memory-Aware Scheduling (MAS)

Memory-aware scheduling considers the effects of memory accesses when ordering the execution of a set of tasks. Memory accesses on one core might incur delays caused by concurrent accesses on another core, because of contention on a shared memory bus. One approach to address this problem is to regulate the rate of off-chip memory references (i.e., those missing in a cache), so that each core cannot exceed a pre-defined threshold [16]. However, there are additional problems that affect the throughput of memory requests. DRAM bank-level parallelism leads to significant variations in the throughput of memory traffic, depending on whether memory accesses are to the same or separate banks [17]. While separate banks are accessible in parallel, requests to the same bank are serialized. Similarly, servicing sequential accesses is faster than servicing random accesses within the same bank, due to row buffering in DRAM. Interleaved accesses to separate rows within the same bank impact row locality, leading to repeated pre-charging of a row buffer. These factors combine to make it difficult to determine the correct memory access rate threshold to avoid excessive bus contention. If we assume separate accesses map to different banks, we may set the threshold too high and it may never be reached even when the bus is heavily contended. Similarly, if we pessimistically assume all accesses are to the same DRAM bank and set the threshold too low, we may trigger memory throttling when the bus is not heavily contended.

A lower memory access rate does not necessarily mean lower contention. Consider the case where two tasks, *task1* and *task2*, are allowed a budgeted number of memory requests every period, T [16]. Figure 2 shows the situation where the tasks run concurrently on separate cores until t, when they exhaust their request budget. Both tasks are then suspended until T. Assuming uniform memory accesses in time, each task reduces its memory access rate by a factor $\frac{T-t}{T}$ in the interval $[0, T]$. However, because the tasks execute at exactly the same time, a reduction in memory access rate does not reduce the contention experienced in the interval $[0, t]$. We call this phenomenon the *Sync Effect*, which occurs when two or more cores have overlapping idle times due to the suspension of tasks. The Sync Effect leads to a drop in CPU and bus utilization without improving task performance.

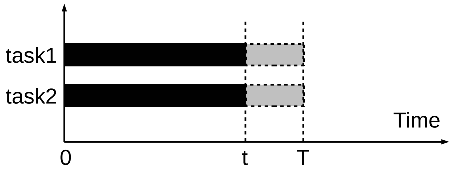

Fig. 2. Sync Effect

To eliminate memory contention requires complete knowledge of access patterns from all cores and Direct Memory Access (DMA) devices, including how they interleave inside the memory controller. Monitoring system wide memory traffic by only looking at each core's requests, either through cache miss events or off-core events [18], is insufficient. This would not detect DMA requests or accesses to a memory domain from a remote node in a Non-Uniform Memory Access (NUMA) system. Fortunately, some multicore architectures, such as the Intel Xeon now provide monitoring events for all types of DRAM traffic.

Our method to deal with memory contention neither relies on memory access rates nor ignores traffic outside cores. It measures memory traffic by looking at the average latency to service memory requests. Unlike a rate-based metric, latency is directly related to application performance. Intel Sandy Bridge and more recent processors provide two uncore performance monitoring events: UNC_ARB_TRK_REQUEST.ALL and UNC_ARB_TRK_OCCUPANCY.ALL. The first event counts all memory requests going to the memory controller request queue ($requests$), and the second one counts cycles weighted by the number of pending requests in the queue ($occupancy$). For example, in Figure 3, request r_1 arrives at time 0 and finishes at time 2. r_2 and r_3 both arrive at time 1 and complete at time 5. At the end of this 5 cycles period, $occupancy = 10$, $requests = 3$. We derive the average latency (cycles) per request as follows:

$$latency = \frac{occupancy}{requests}$$

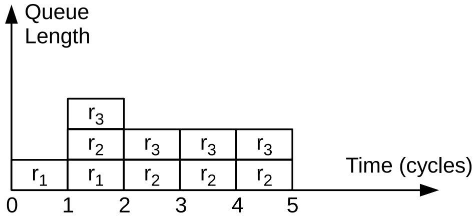

Fig. 3. Example of Memory Controller Occupancy and Requests

MARACAS is configured with a request latency threshold, MAX_MEM_LAT. The threshold is global rather than per-core for comparison with the observed overall bus traffic. Memory throttling commences when the observed average latency exceeds or equals MAX_MEM_LAT, as shown in Algorithm 1 (line 9). A memory monitoring thread assigned to a dedicated VCPU periodically updates the average latency via a *MONITOR* procedure. The period is set to a constant MEM_PERIOD.

When throttling is applied in MARACAS, background scheduling on the corresponding core is temporarily disabled, and the core goes idle. This reduces contention on the memory controller, shared cache lines and Miss Status Holding Registers [19]. While the Sync Effect is still possible, MARACAS is able to detect the contention and apply further throttling. When one or more cores throttle their usage of background time, other cores in foreground mode are able to make greater progress due to the reduced contention.

Instead of simply disabling the cores with the most traffic, we adopt a proportional throttling scheme. Suppose the ith core in a set of n cores generates m_i requests to the memory controller in time t_i, which yields a memory access rate $r_i = \frac{m_i}{t_i}$. Larger values of r_i cause a greater degree of memory throttling on the ith core. Global variable $num\_throttle$ is used to tell the scheduling sub-system how many cores (referred to as *cpus* in Algorithm 1) need to be throttled. When the core-local scheduler is switched to background mode, it calls function *IS_BG_SCHED* , which returns `TRUE` if a task is able to run. `count` keeps track of how many cores should be allowed to run in background mode if the current core is allowed to do so as well. $bg\_vtime[i]$ on core i is the product of the background execution time consumed in the current period (MEM_PERIOD) and the weight, $mem\_weight[i]$, which is generated inside *CALC_WEIGHTS*. Higher values of $bg\_vtime[i]$ relative to those on other cores increase the likelihood of core i being throttled.

Once memory throttling is activated, it begs the question how long it should be applied. While average memory request latency is used to determine bus contention, it is not necessarily the best metric to identify a reduction in memory bandwidth demand. This is because a reduction in memory access latency could be due to throttling background time, rather than a drop in memory requests from the running VCPUs. Let $R_{cur} = \sum_{i=1}^{n} m_i$ in the current period [4], and R_{high} be the largest R_{cur} since throttling began. In MARACAS, if $R_{cur} <= R_{high} \times$IDLE_MEM (IDLE_MEM is a configurable parameter between 0 and 1), the system-wide memory access intensity is considered to be lower than before and throttling is reduced gradually (by decreasing $num\_throttle$, Algorithm 1 line 14).

V. Multicore VCPU Scheduling

MARACAS is built on Quest's VCPU scheduling framework. Special kernel threads, associated with dedicated migration VCPUs, are responsible for the movement of VCPUs between cores. Every core has one migration thread, but only one can be active at a certain time while others are blocked. A migration thread is responsible for checking the load of every core and deciding whether to perform load balancing. If a migration thread decides to move a VCPU, it must first identify a destination core. A VCPU is only migrated if it passes a schedulability test for the destination core. The cost of the test, which amounts to a relatively simple utilization bound calculation, is factored into the migration thread's budget. A migration thread is woken up when there is a scheduling event (e.g., a task blocks, wakes up or terminates), at which point it performs a rebalance check and potential VCPU migration. A migration thread is not woken up by the periodic sleeping of real-time tasks, which are ineligible to run in foreground mode until their budgets are replenished. Currently, only one VCPU is allowed to be migrated within one period of the migration thread. This is purely a policy decision and not an inherent design limitation. A pending event counter records the number of scheduling events that happen before the active migration thread completes the transfer of a VCPU.

For real-time systems, the migration process has to be predictable. Care must be taken to make sure migration costs do not impact the timing requirements of the VCPU being relocated to another core. First, migration threads are set to the highest priority on their respective cores to avoid preemption during the migration process. Second, each VCPU that is created must pass a schedulability test on its assigned core. This means the migration thread's execution of its entire budget C_m does not lead to any other local VCPUs missing their deadlines. Therefore, as long as the migration cost is smaller than C_m, timing constraints on the local core will not be violated. Finally, the migrated VCPU must pass a schedulability test at the destination to ensure that it does not violate any timing guarantees on that core.

[4] Or, equivalently, $R_{cur} = \sum_{i=1}^{n} r_i$ over a unit-length period.

Algorithm 1 Memory-Aware Scheduling

```
procedure MONITOR
    /* update all $m_i$ */
    /* clear all $bg\_vtime[i]$ */
    /* UNC_ARB_TRK_REQUEST.ALL */
    $requests = get\_requests()$
    /* UNC_ARB_TRK_OCCUPANCY.ALL */
    $occupancy = get\_occupancy()$
    $latency = occupancy/requests$
    if $latency >= MAX\_MEM\_LAT$ and
        $num\_throttle < num\_cpus$ then
        $num\_throttle++$
    else if $IS\_LESS\_TRAFFIC()$ and
        $num\_throttle > 0$ then
        $num\_throttle--$
    end if
    if $num\_throttle > 0$ then
        $CALC\_WEIGHTS()$
    end if
end procedure

procedure IS_LESS_TRAFFIC
    if $R_{cur} <= R_{high} \times IDLE\_MEM$ then
        return TRUE
    else
        return FALSE
    end if
end procedure

procedure CALC_WEIGHTS
    for all $cpu$ do
        $mem\_weight[cpu] = m_{cpu}/R_{cur}$
    end for
end procedure

procedure IS_BG_SCHED
    if $num\_throttle <= 0$ then
        return TRUE
    end if
    if $num\_throttle >= num\_cpus$ then
        return FALSE
    end if
    $count = 0$
    $self = get\_local\_cpu\_id()$
    for all $cpu$ do
        if $cpu\ != \ self$ and
         $bg\_vtime[cpu] <= bg\_vtime[self]$ then
            $count++$
        end if
    end for
    if $count < num\_cpus - num\_throttle$ then
        return TRUE
    else
        return FALSE
    end if
end procedure
```

In MARACAS, the VCPU migration process is as simple as locking two runqueues (on the source and destination cores), detaching the VCPU data structure from the source queue and attaching it to the destination. Memory address space copying is not needed during migration unless cache partitioning is enabled within the underlying Quest system. Quest runs as a single system image across all cores, and memory copying during migration is only necessary if page color-aware cache partitioning is configured by a system designer, who wishes to have stronger resource isolation.

Let E_{lock} be the overhead of locking a runqueue, and E_{struct} be the overhead of moving a VCPU data structure in the worst case. Then, the following condition must hold:

$$C_m \geq 2 \times E_{lock} + E_{struct}$$

If cache partitioning is enabled, then pages of a migrated address space need recoloring on the destination core. MARACAS builds upon the Quest system guarantee that process address spaces are limited to a maximum size. Hence, it is possible to place an upper bound on the memory copying overheads. Let E_{page} be the cost of copying one page, and P_{max} be the maximum number of pages. Then, the new migration constraint is:

$$C_m \geq 2 \times E_{lock} + E_{struct} + P_{max} \times E_{page}$$

To keep migration costs down, it makes sense to reduce the frequency with which migrations occur. While MARACAS currently invokes a migration thread on the local core every time a scheduling event occurs, it is possible to define a minimum time interval (or maximum frequency) in which migration is allowed. It is also the case that C_m should be set sufficiently large to encompass the cost of migration in foreground mode. This way, the migration thread itself will not be throttled if congestion is detected. Further studies of the exact costs of migration, and policies to control the migration frequency, are left to future work.

A. VCPU Load Balancing (VLB)

In Linux, the CPU load is defined as the sum of all local tasks' scheduling weights, where a weight is decided by a task's priority. Every core periodically runs a load balancing algorithm, which attempts to minimize the difference in load amongst all cores. The goal is to let every task of the same priority have the same amount of CPU time.

Load balancing in our real-time VCPU framework differs from that in a general-purpose OS, as it deals with VCPUs that have CPU reservations. Each VCPU is guaranteed its CPU reservation irrespective of the mapping of VCPUs to cores. Balancing VCPUs so they received the same amount of CPU time would penalize those with larger reservations. VCPUs with larger budgets in a given period of time would have less background time than those with smaller reservations. In observance of these differences, we propose an alternative method of load balancing for a system of VCPUs on a multicore platform.

Let each VCPU, V_i, have a utilization factor $U_i = \frac{C_i}{T_i}$. We then define the Slack-Per-VCPU (SPV) of a core as $\frac{1.0-\sum_{i=1}^{v} U_i}{v}$, where v is the number of VCPUs on the corresponding core. The smaller the SPV is, the heavier the load is for the core. For load balancing, we attempt to equalize the SPV values across cores.

Algorithm 2 is the main body of VCPU load balancing scheme. When any VCPU blocks, the kernel tries to find a core with the smallest SPV value and activates its corresponding migration thread. Whenever a VCPU is awoken, the migration thread on the same core is activated as well. Every migration thread runs procedure *REBALANCE* when active, which migrates VCPUs from the current core to the one that has the most idle time, as indicated by the largest SPV. *FIND_HOST_CPU* identifies the core with the largest idle time that can feasibly schedule a new VCPU. For a feasible schedule, all VCPUs must satisfy their foreground scheduling requirements on the given core. Line 18 starts by finding a target core for the VCPU with the lowest utilization on the local (source) core. If a core exists, lines 29 onwards check to see if an alternative VCPU could be migrated to the target core to reduce the SPV imbalance between the source and destination. By checking the feasibility of migrating the lowest utilization VCPU first, we avoid attempting to reduce the SPV imbalance across cores for higher utilization VCPUs that would not be schedulable at the destination. Finally, line 48 is the condition to terminate the rebalancing procedure.

Algorithm 2 VCPU Load Balancing

```
 1: procedure FIND_HOST_CPU($new\_vcpu$)
 2:     $max = 0$
 3:     for all $cpu$ do
 4:         if $schedulability\_test(cpu, new\_vcpu) == FALSE$ then
 5:             continue
 6:         end if
 7:         if $SPV(cpu) > max$ then
 8:             $max = SPV(cpu)$
 9:             $host = cpu$
10:         end if
11:     end for
12:     return $host$
13: end procedure

14: procedure REBALANCE
15:     $src\_cpu = current\_cpu\_id()$
16:     /* return the VCPU with the smallest utilization on a core */
17:     $min\_v = get\_smallest\_ut\_vcpu(src\_cpu)$
18:     $dst\_cpu = FIND\_HOST\_CPU(min\_v)$
19:     if $dst\_cpu == src\_cpu$ then
20:         return
21:     end if
22:     $src\_spv = get\_SPV\_of(src\_cpu)$
23:     $dst\_spv = get\_SPV\_of(dst\_cpu)$
24:     if $runqueue\_length(dst\_cpu) == 0$ then
25:         $imbalance = \infty$
26:     else
27:         $imbalance = dst\_spv - src\_spv$
28:     end if

29:     for all $vcpu$ in $runqueue(src\_cpu)$ do
30:         if $runqueue\_length(src\_cpu) <= 1$ then
31:             break
32:         end if
33:         /* calculate a core's new SPV as if a VCPU is added */
34:         /* without actually adding VCPU to the core */
35:         $dst\_spv = get\_SPV\_add\_one(dst\_cpu, vcpu)$
36:         /* calculate a core's new SPV as if a VCPU is removed */
37:         /* without actually removing VCPU from the core */
38:         $src\_spv = get\_SPV\_remove\_one(src\_cpu, vcpu)$
39:         if $dst\_spv < src\_spv$ and
40:                 $src\_spv - dst\_spv >= imbalance$ then
41:             continue
42:         end if
43:         if $schedulability\_test(dst\_cpu, vcpu) == FALSE$ then
44:             continue
45:         end if

46:         $move\_vcpu(src\_cpu, dst\_cpu, vcpu)$
47:         $imbalance = dst\_spv - src\_spv$
48:         if $imbalance <= 0$ then
49:             break
50:         end if
51:     end for
52: end procedure
```

B. Cache-Aware Scheduling (CAS)

To improve performance isolation and timing predictability, we extended the VLB algorithm to work with the cache partitioning sub-system in Quest. In the current implementation, static cache partitioning is used and every core is assigned some fixed number of colors during system initialization. Tasks running on a core are allocated page frames whose colors are restricted to the set reserved for the corresponding core. The VCPU creation API in Quest was modified to allow tasks to specify the minimum number of page colors needed for their cache requirement:

```
bool vcpu_create(uint C, uint T, uint colors);
```

CAS is similar to VLB except that before a VCPU is migrated the destination core is checked to see if it has sufficient page colors available. Migration only takes place if there are enough page colors to meet the VCPU's cache requirement. The migration process takes longer since the task's address space associated with the migrating VCPU has to be recolored [12]. As a result, migration threads need larger CPU reservations.

Although we focus on static page coloring in this paper, we have also studied dynamic page coloring. Our experiences suggest that dynamic page coloring incurs too much timing unpredictability to be appropriate for hard real-time tasks, but could be beneficial for non-real-time or low-criticality tasks in a mixed criticality system.

VI. Evaluation

We evaluated the MARACAS multicore scheduling framework in Quest, using the hardware platform in Table I.

Processor	Intel Core i5-2500k quad-core
Caches	6MB L3 cache, 12-way set associative, 4 cache slices
Memory	8GB 1333MHz DDR3, 1 channel, 2 ranks, 8KB row buffers

TABLE I
Hardware Specification

A. Background Scheduling

The first experiment investigated the effectiveness of background scheduling using two test cases. In the first case (*vcpu + bg*), tasks were run with background scheduling enabled, while in the second case (*vcpu*), background scheduling was disabled. Four Mälardalen benchmarks [20] (*compress*, *adpcm*, *fir*, and *matmult*) were started simultaneously on the same core. Every task was assigned a VCPU with the same capacity C (ms) and a fixed period of $T = 100$ ms. Unless stated otherwise, the value $T = 100$ ms was used throughout the experimental evaluations in this paper. In this experiment, benchmarks were executed for 5 minutes, after which the counts of their instructions retired were collected. Figure 4 shows that background scheduling improves progress in every case. The total instructions retired are approximately equal for each benchmark with different values of C. Greater values of C increase the base level instructions retired when a VCPU obtains its guaranteed share of the CPU.

B. Memory-Aware Scheduling

This section compares Memory-Aware Scheduling (MAS) using our latency metric against an approach using a rate metric. For this experiment, we developed a memory-intensive benchmark, *m_jump* (with pseudocode shown in Code 1), that operates on a 6 MB data array, which is large enough to span the entire last-level cache (L3). The benchmark writes to the first 4 of every 64 bytes in the array. As every cache line is also 64 bytes, this causes the entire cache to be filled. After every write, *m_jump* jumps 8 KB forward in order to avoid cache prefetching effects. It is worth noting that caches cannot be disabled for this experiment, even though our focus is on memory performance. If caches were disabled, every instruction would be fetched from memory. This would effectively force CPUs to run at the same speed as the memory bus, reducing the likelihood of bus congestion.

Code 1. *m_jump*

```
byte array[6M];
for (uint32 j = 0; j < 8192; j += 64)
    for (uint32 i = j; i < 6M; i += 8192)
        <Variable delay added here>
        array[i] = i;
```

To establish a fair comparison between the rate and latency metrics, we first performed several profiling experiments. Three *m_jump* tasks (*task1*, *task2* and *task3*) were executed on separate cores for 5 minutes *without memory throttling*. The task parameters, (C, T), were set to $(20, 40)$, $(25, 50)$ and $(30, 60)$, respectively. In each run, we inserted a time delay between memory accesses in the *m_jump* code for *task1* and *task2*, by performing multiplication operations on a register value for a variable number of iterations. The use of a register was to avoid any extra memory requests that might affect the experiment. At the end of the experiment, we recorded the total system-wide bus traffic, average memory request latency and *task3*'s instructions retired in foreground mode. Results are shown in Figure 5. The *Bus Traffic* curve shows data points for the memory latency X and corresponding traffic Y. Matching X and Y data points on the *Bus Traffic* curve are used to establish latency and rate thresholds, respectively, for memory throttling. Derivation of these thresholds is described later. The corresponding *Instructions* curve enables thresholds to be set that trade-off performance of the target application (*task3*) and the entire system memory throughput. Notice that MAS does not require performance profiling to function properly. Profiling is used here to establish comparable thresholds for the two memory throttling metrics.

From Figure 5, we chose three data points on the *Bus Traffic* curve that straddled the intersection with the *Instructions* curve. The chosen values represent several cases when the bus traffic is rising to its limit. The latencies for these three points were 157, 183 and 228 cycles, respectively, as shown by the vertical lines. For each latency, we also recorded the foreground performance of *task3* on the *Instructions* curve, which resulted in three experimental configurations, E1, E2 and E3 (See Table II).

	Bus Traffic (GB)	Latency	*task3* Instructions Retired ($\times 10^8$)
E1	1128	228	249
E2	1049	183	304
E3	976	157	357

TABLE II
PROFILE CONFIGURATIONS

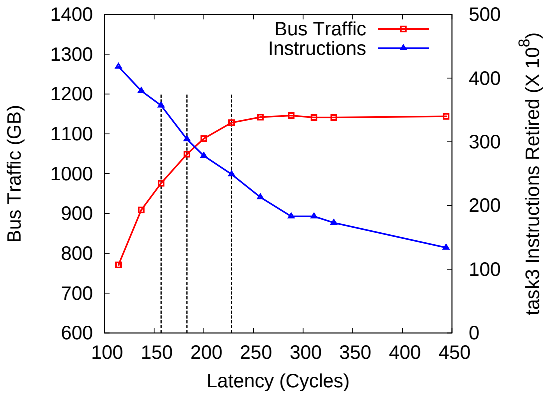

Fig. 5. Bus Traffic & Instructions Retired versus Latency

The values in the *Latency* column were used as thresholds for *latency-based memory throttling*. Data in the *Bus Traffic* column shows the gigabytes transferred across the memory bus in a 5 minute interval. We converted these values into a memory service rate per MEM_PERIOD (set to 2 seconds), to establish comparative thresholds for *rate-based memory throttling*. MEM_PERIOD is set empirically, with smaller values enabling finer-grained monitoring of bus traffic and larger values imposing lower system overhead on memory-aware scheduling. We chose a value of MEM_PERIOD=2s as a reasonable trade-off when considering task scheduling periods in milliseconds. The last column in Table II serves as a reference, showing the expected performance of *task3* using the corresponding thresholds.

Next, we repeated the previous experiment *with memory throttling*. A fixed delay was added to the *m_jump* code of *task1* and *task2* so they would cause heavy bus contention. With each configuration (E1, E2, E3), we compared rate- and latency-based memory throttling. Figure 6 shows the resultant foreground performance of *task3*. In both E1 and E2 cases, our latency-based throttling approach was able to reduce bus contention so that the target application's performance was better than expected. In contrast, the rate-based approach failed to achieve the expected performance of *task3*. In case E3, the latency threshold was too low, leading to insufficient background time (BGT) to reduce bus contention. However, latency-based throttling still enabled *task3* to execute more instructions (and, hence, make further progress) than the rate-based approach.

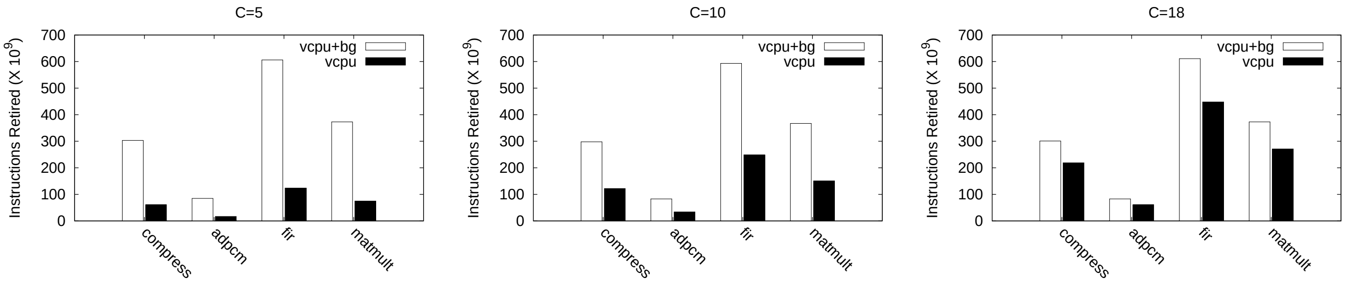

Fig. 4. Background Scheduling

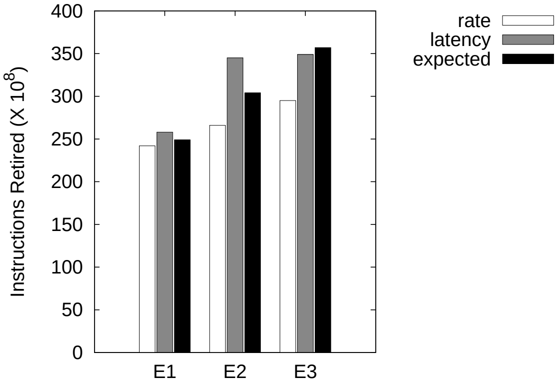

Fig. 6. Comparison between Rate- and Latency-based Throttling

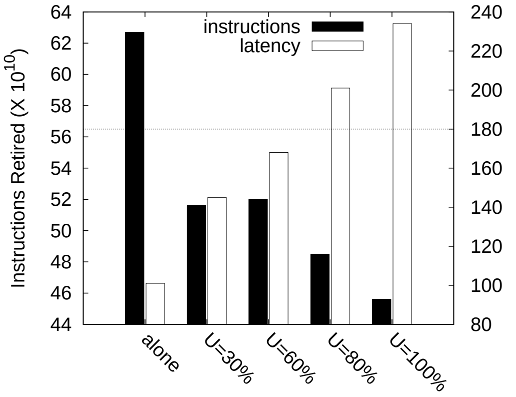

Fig. 7. Foreground Performance of Canny

The E3 case reveals a limitation of MARACAS: the effectiveness of memory throttling depends on the amount of BGT for each core. We demonstrated this dependence through another experiment. A *canny* edge detection benchmark used in image processing was executed for 10 minutes on a VCPU with parameters $C = 50$ ms, $T = 100$ ms. During this time, *canny* repeatedly processed a 720×480 pixel image on a single core. Three *m_jump* benchmarks were executed on the other three cores, with their VCPU periods set to $T = 150$, 100 and 50 ms, respectively. Different T values were used to avoid the Sync Effect described in Section IV. The foreground utilizations of the VCPUs associated with the *m_jump* benchmarks were varied to yield five different cases in this experiment. For cases, *U=30%*, *U=60%*, *U=80%* and *U=100%*, each *m_jump* VCPU was allocated 30, 60, 80 and 100% CPU utilization, respectively. For the special case *alone*, *canny* was executed without any *m_jump* co-runners.

Figure 7 shows the performance of *canny* in foreground mode. For *m_jump* utilizations below 80%, the system was able to maintain memory request latency below the threshold, MAX_MEM_LAT = 180 cycles. As the *m_jump* utilization increased, MARACAS would gradually lose its capability to provide service quality to *canny*.

C. VCPU Load Balancing

With VLB, every task has a fair share of BGT. This is shown by the following experiment comprising two groups. A *static* group used only the *FIND_HOST_CPU* procedure from VLB (Algorithm 2) and statically mapped tasks to cores. A second *VLB* group used the complete VLB algorithm, which allowed threads to be migrated between cores. In each case, we created 16 instances of the *compress* benchmark that were started at 1 second intervals. Every benchmark was assigned a VCPU capacity, C, and duration D. The duration specified how long the task executed in minutes. In all cases, the VCPU periods assigned to each benchmark were set to $T = 100$ ms.

We generated 10 sets ($k0$ to $k9$) of parameter values for each group of 16 tasks. Parameters C and D were generated from a uniform random distribution over the range $1 - 14$ ms and $2 - 11$ minutes, respectively. The range of C values caused variation in the foreground utilization of each VCPU, while ensuring the total utilization remained below the RMS bound [8]. The range of D values ensured that within a 10 minute monitoring period the system load was dynamic: some tasks terminated while others remained active. Each of the 16 tasks in the same experimental group were assigned a randomly chosen value of C, while the first 14 tasks were assigned randomly chosen values of D. Two other tasks in each group (*task1* and *task2*) were executed for the full duration of each experimental run. The total instructions retired in *background* mode were recorded for *task1* and *task2* over a 10 minute interval from when all 16 tasks were first assigned to cores.

Figure 8 shows the result of VCPU load balancing. In the *static* case, *task1* and *task2* exhibit highly variable progress

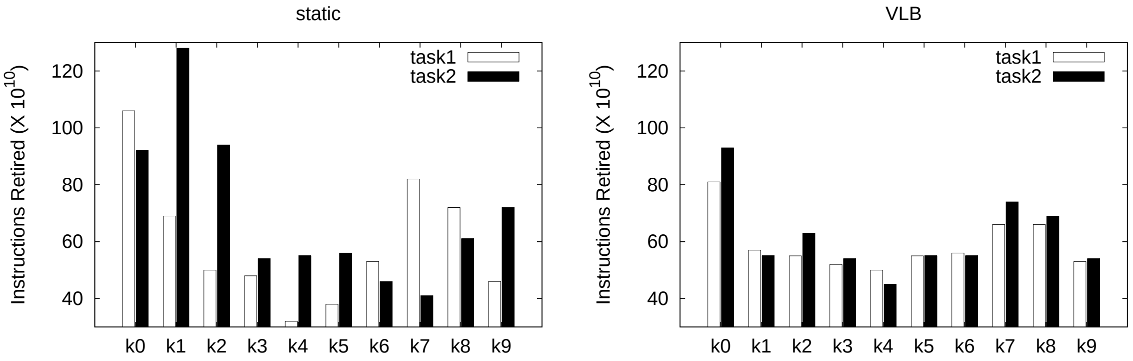

Fig. 8. Instructions Retired in Background Mode

across the 10 parameter sets. In contrast, the VLB case shows that dynamic load balancing achieves more evenly distributed progress for the two observed tasks. This suggests that VLB is more effective at distributing background CPU time equally.

D. Cache-Aware Scheduling

For the next experiment, we set the memory pool inside MARACAS' cache-aware memory allocator to be 1 GB. The hardware specification in Table I allows a total of 32 allocatable page colors [12]. We devised several programs to observe the effects of shared caches on task execution. Our *mwalk* program writes to elements of a 1 MB array in a pseudo-random order to eliminate the benefits of hardware cache prefetching. Another *hog* program repeatedly scans a 2 MB array of integer elements in a sequential order.

We first started 15 *hog* tasks, each with parameters C and D, at 1 second intervals. Values for C and D were randomly generated in the same way as for the previous VLB experiments. After the last *hog* was activated, we executed an instance of *mwalk* for 10 minutes on a VCPU with $C = 6$ ms and $T = 100$ ms. The last-level cache (LLC) miss ratio in misses per reference was then recorded for *mwalk*'s execution.

The experiment above was repeated 10 times with different random sets of parameters ($t0$ to $t9$) applied to the 15 instances of *hog*. Three cases were considered: *share*, *c0* and *c6*. In the *share* case, the VLB algorithm was tested on a Quest system without page coloring. In the *c0* and *c6* cases, VLB was used with page coloring to implement cache-aware scheduling (CAS), with the LLC partitioned in the page color ratio $4:4:10:14$ across the four cores. The *mwalk* task requested 0 page colors for *c0*, and 6 page colors for *c6*. *c0* represents the case where there is no cache space *reserved* for the task, even though it will obviously need memory pages for its address space.

Figure 9 shows the LLC Miss Ratio for *mwalk* in all 10 experimental runs. In the presence of inter-core cache interference, *mwalk* suffers a very high cache miss ratio as seen by the *share* cases. Even with cache partitioning, the *c0* case does not always perform well (e.g., for experimental runs $t3$, $t4$ and $t6$). This happens because a migration thread may place *mwalk* on a core with a cache partition that is smaller than its working set, causing self-conflict misses. In general, the *c6* case performs best, although cache misses still occur due to context-switching between *mwalk* and other tasks on the same core. When a new task executes on a given core, it may evict cache lines for the previously running task. As stated earlier, MARACAS has the option of establishing a fixed number of page colors for each core, ensuring cache partitioning at the core-level. It is still possible for tasks on the same core to be assigned overlapping page colors. However, given sufficient colors, tasks can be allocated memory pages from a separate set of page colors.

To show the effects of context-switching between tasks on the same core, we ran another set of experiments similar to the above. This time, we varied the working set of *mwalk* using three different array sizes of *0.5MB*, *1MB* and *2MB*. In each case, the cache requirement to accommodate the working set of *mwalk* was passed to the Quest kernel. Figure 10 shows that for a larger working set, the task may consume all its VCPU budget before scanning the entire array, only to see its cache contents evicted by another task before resuming execution.

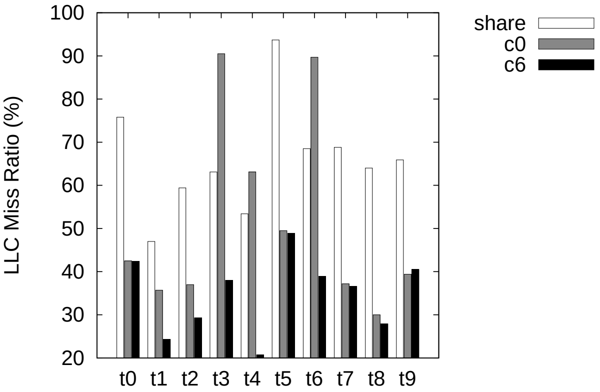

Fig. 9. CAS Evaluation

VII. Related Work

A. Multicore Real-Time Scheduling

Prior work on multicore real-time scheduling has predominantly focused on *global* [21], [22] and *partitioned* [23], [24] approaches. Global scheduling selects tasks from a system-wide run queue and allows for task migrations between cores. Partitioned scheduling statically assigns each task to

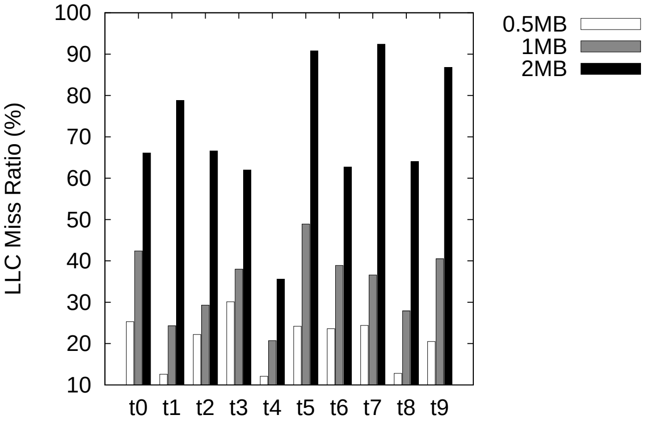

Fig. 10. *mwalk* With Different Working Set Size

a core, where it is scheduled from a local run queue. Partitioned approaches take advantage of well-studied uniprocessor scheduling techniques [25], [26], while global approaches tend to achieve better CPU utilization across the whole system [27]. To improve utilization, while avoiding the overheads of global scheduling, researchers have now developed *semi-partitioned* schemes. Examples include EDF-fm [28] and EDF-WM [29]. Semi-partitioned scheduling allows for a subset of tasks to migrate, while others remain statically mapped to cores.

The MARACAS scheduling framework built on Quest implements a local scheduler for each core, but allows for tasks and their VCPUs to be migrated to other cores. The system avoids the need for a global scheduling queue by allowing each local scheduler to access load information for remote cores, as part of dynamic load balancing. The redistribution of tasks and VCPUs in our work is intended to balance the background CPU time on each core, for use in the prevention of cache and memory bus contention.

MARACAS' use of surplus CPU time contrasts with slack reclamation algorithms. MARACAS regulates the use of slack time, beyond that reserved by VCPUs, to address both contention and improve task progress. Slack reclamation algorithms [15] typically allow aperiodic or low-priority jobs to execute whenever high-priority ones may be safely postponed. Approaches such as CBS [30], CASH [31], GRUB [32] and BACKSLASH [33] apply techniques to reclaim *dynamic* slack, which is unused but reserved capacity. They allow for tasks to acquire early access to surplus resources. MARACAS focuses on the redistribution of static slack (i.e., unreserved capacity) across cores, to maximize the number of cores that can be throttled when there is resource contention.

B. Shared Resource Management

The effects of shared caches, buses and DRAM banks on task execution have been studied in recent years. Page coloring [34], [35], [11], [36] is a commonly used technique to partition physically-indexed caches on multicore processors. In 2006, Cho and Jin [37] applied page coloring to a multicore system, with the goal to place data in cache slices that are closer to the CPU running the target application. Tam et al [38] implemented static page coloring in a prototype Linux system, which demonstrated improved performance by reducing cache contention amongst cores. SRM-Buffer [39] used page coloring to limit the cache space accessible to a system page cache, resulting in reduced cache interference from file operations. COLORIS [12] demonstrated an efficient method for dynamic page coloring, although it was primarily intended for improved system performance rather than timing predictability.

A number of other researchers have also looked at real-time cache-aware resource management. This includes work on page coloring with cache lockdown for use in mixed criticality systems [40]. Ward et al [41] studied cache locking and scheduling techniques, to reduce worst-case execution times (WCETs) of higher-criticality hard real-time tasks in the presence of lower-criticality soft real-time tasks. Calandrino et al [42] studied several real-time cache-aware scheduling policies based on the cache utilization of multi-threaded tasks. Metrics such as the working set size were used to establish a utilization threshold which, when reached, would trigger a cache-aware policy to select a task based on under-utilized cache space and available cores. Kim et al [43] developed an OS-level cache management scheme for multicore real-time systems, using Linux/RK. The work included the development of a response time schedulability test for tasks that share cache partitions. Mancuso et al [44] also developed a framework to analyze and profile task memory access patterns, including a kernel-level cache management scheme to enforce deterministic cache allocations for the most frequently accessed memory areas. These works mostly complement MARACAS, which uses page coloring-aware techniques to partition shared caches and determine the assignment of tasks to cores.

Bellosa et al [45] developed a memory throttling technique using hardware performance counters to determine memory bus usage. More recently, MemGuard [16] was developed to address timing variations caused by memory references from different cores. Each core is assigned a memory budget, which limits the number of memory accesses in a specified interval. To improve bandwidth utilization, MemGuard predicts the actual bandwidth usage of each core in the upcoming period. For cores that do not use all their budgets, they contribute their surplus to a global pool, which is shared amongst all cores. A similar user-space technique [46] was developed to allow memory to be budgeted to individual, or groups of, tasks. With MemGuard, mispredictions in memory bandwidth usage may lead to one core donating too much budget to achieve its minimum guarantee. That said, MARACAS could use similar ideas to MemGuard to ensure a minimum memory bandwidth guarantee for tasks running in foreground mode.

PALLOC uses a DRAM bank-aware buddy allocator to assign page frames to applications so that bank-level contention is avoided [17]. Both MemGuard and PALLOC are part of an effort to develop a Single Core Equivalence (SCE) framework [47]. SCE attempts to treat each core in a multicore processor as if it were a separate chip, to ensure that a task's worst-case execution time (WCET) is not affected by other tasks running on different cores. MARACAS does not focus on total isolation between cores, but instead is aimed at improving the progress of co-running workloads beyond their baseline timing guarantees.

Finally, Dirigent [48] is a system that regulates the progress of latency-constrained (*foreground*) tasks in the presence of non-time-constrained (*background*) tasks. Dirigent reduces the performance variation of foreground applications caused by memory contention, while maintaining a high throughput for background tasks. The system works by first offline profiling the execution of latency-constrained tasks when running alone. An online execution time predictor and controller then adjust resources available to foreground tasks, to ensure their latency constraints in the presence of contention from background tasks. This is similar to how MARACAS throttles background cycles for some workloads, while others are allowed to execute. However, MARACAS applies throttling when it determines the likelihood of congestion, according to a latency-based memory request metric. Dirigent adjusts resource allocations according to the online progress of tasks compared to their offline execution.

VIII. Conclusions and Future Work

This paper describes a real-time multicore scheduling framework called MARACAS. MARACAS is a memory-aware, real-time-aware and cache-aware scheduling subsystem in the Quest operating system. It builds upon Quest's real-time VCPU scheduling infrastructure that was described in earlier work for uniprocessors. In this paper, we focus on VCPUs that operate as Sporadic Servers, each having a time budget and period. The MARACAS framework takes advantage of surplus CPU cycles on each core, after meeting the foreground timing requirements of each VCPU, to improve system performance.

We show how to load balance VCPUs across cores to both guarantee VCPU timing requirements and evenly distribute surplus CPU cycles. MARACAS recolors address spaces associated with migrating VCPUs to avoid cache contention and maintain performance isolation. Of particular significance is MARACAS' ability to throttle memory accesses by carefully regulating the amount of surplus, or background, CPU time available to cores. When there is heavy memory bus contention, each core will stop the allocation of background CPU time, and will instead limit CPU cycles to only those necessary to meet VCPU timing constraints. MARACAS uses hardware performance counters to derive an *average memory request latency* metric. This metric is used to determine system-wide memory traffic congestion and to control the surplus CPU time available on each core. This approach is shown to outperform alternative techniques that limit the rate of memory accesses to avoid bus contention.

MARACAS is a soft real-time framework that unifies the management of multiple contended resources, which affect timing guarantees. It targets applications that improve their quality when given more execution time, such as in data sampling, numerical integration and imprecise computations. Background scheduling provides a way to improve task progress while reducing co-runner contention. The rebalancing of background time enables as many cores as possible to throttle their execution and therefore reduce resource contention.

As part of future work, we plan to extend our memory throttling mechanism to handle DMA-related traffic. We will also investigate the performance and predictability of MARACAS in the presence of IO activities. While MARACAS uses foreground time to make baseline guarantees and background time to improve quality of service, we plan to compare against techniques that have no such time separation. The source code for MARACAS is available upon request.

Acknowledgment

This work is supported by the National Science Foundation under Grant # 1527050. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We also thank the anonymous reviewers who have helped improve this paper, and Intel Corporation for its generous support.

References

[1] M. Danish, Y. Li, and R. West, "Virtual-CPU Scheduling in the Quest Operating System," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '11, 2011, pp. 169–179.

[2] A. Quagli, D. Fontanelli, L. Greco, L. Palopoli, and A. Bicchi, "Designing Real-Iime Embedded Controllers using the Anytime Computing Paradigm," in *Proceedings of the 14th IEEE International Conference on Emerging Technologies and Factory Automation*, 2009, pp. 929–936.

[3] J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. s. Yu, J. Y. Chung, and W. Zhao, "Algorithms for Scheduling Imprecise Computations," *Computer*, vol. 24, no. 5, pp. 58–68, May 1991.

[4] G. Banga, P. Druschel, and J. C. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[5] C. W. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994, pp. 90–99.

[6] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems," *Real-Time Systems Journal*, vol. 1, no. 1, pp. 27–60, 1989.

[7] M. Stanovich, T. P. Baker, A. I. Wang, and M. G. Harbour, "Defects of the POSIX Sporadic Server and How to Correct Them," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

[8] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[9] R. West, P. Zaroo, C. A. Waldspurger, and X. Zhang, "Online Cache Modeling for Commodity Multicore Processors," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 4, pp. 19–29, Dec. 2010.

[10] ——, "CAFÉ: Cache-Aware Fair and Efficient Scheduling for CMPs," in *Multicore Technology: Architecture, Reconfiguration and Modeling*. CRC Press, 2013.

[11] J. Liedtke, H. Härtig, and M. Hohmuth, "OS-Controlled Cache Predictability for Real-Time Systems," in *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1997.

[12] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A Dynamic Cache Partitioning System Using Page Coloring," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. ACM, 2014, pp. 381–392.

[13] S. Blagodurov, S. Zhuravlev, and A. Fedorova, “Contention-Aware Scheduling on Multicore Systems,” *ACM Transactions on Computer Systems*, vol. 28, no. 4, December 2010.

[14] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, “Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems,” in *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, 1998, pp. 150–164.

[15] J. P. Lehoczky and S. Ramos-Thuel, “An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems,” in *Proceedings of the IEEE Real-Time Systems Symposium*, December 1992, pp. 110–123.

[16] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-core Platforms,” in *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, ser. RTAS ’13, 2013, pp. 55–64.

[17] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, “PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms,” in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS ’14, 2014.

[18] *Intel 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes 3A, 3B, 3C, and 3D: System Programming Guide*, Intel, 2014.

[19] P. K. Valsan, H. Yun, and F. Farshchi, “Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems,” in *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS ’16, 2016.

[20] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, “The Mälardalen WCET Benchmarks – Past, Present and Future,” in *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, B. Lisper, Ed. OCG, Jul. 2010, pp. 137–147.

[21] S. K. Dhall and C. L. Liu, “On a Real-Time Scheduling Problem,” *Oper. Res.*, vol. 26, no. 1, pp. 127–140, Feb. 1978.

[22] B. B. Brandenburg and J. H. Anderson, “On the Implementation of Global Real-Time Schedulers,” in *Proceedings of the 30th IEEE Real-Time Systems Symposium*, ser. RTSS ’09, 2009, pp. 214–224.

[23] S. Baruah and N. Fisher, “The Partitioned Multiprocessor Scheduling of Sporadic Task Systems,” in *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, ser. RTSS ’05, 2005, pp. 321–329.

[24] ——, “The Partitioned Multiprocessor Scheduling of Deadline-constrained Sporadic Task Systems,” *IEEE Transactions on Computers*, vol. 55, no. 7, pp. 918–923, July 2006.

[25] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia, “Worst-case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems,” in *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, ser. ECRTS ’00, 2000, pp. 25–33.

[26] J. M. Lopez, J. L. Diaz, and D. F. Garcia, “Minimum and Maximum Utilization Bounds for Multiprocessor RM Scheduling,” in *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, ser. ECRTS ’01, 2001, pp. 67–75.

[27] R. I. Davis and A. Burns, “A Survey of Hard Real-time Scheduling for Multiprocessor Systems,” *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, Oct. 2011.

[28] J. H. Anderson, V. Bud, and U. C. Devi, “An EDF-based Restricted-migration Scheduling Algorithm for Multiprocessor Soft Real-Time Systems,” *Real-Time Syst.*, vol. 38, no. 2, pp. 85–131, Feb. 2008.

[29] S. Kato, N. Yamasaki, and Y. Ishikawa, “Semi-partitioned Scheduling of Sporadic Task Systems on Multiprocessors,” in *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, July 2009, pp. 249–258.

[30] L. Abeni, G. Buttazzo, S. Superiore, and S. Anna, “Integrating Multimedia Applications in Hard Real-Time Systems,” in *Proceedings of the 19th IEEE Real-time Systems Symposium*, 1998, pp. 4–13.

[31] M. Caccamo, G. Buttazzo, and L. Sha, “Capacity Sharing for Overrun Control,” in *Proceedings of the 21th IEEE Real-Time Systems Symposium*, December 2000, pp. 295–304.

[32] G. Lipari and S. Baruah, “Greedy Reclamation of Unused Bandwidth in Constant-Bandwidth Servers,” in *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, June 2000, pp. 193–200.

[33] C. Lin and S. A. Brandt, “Improving Soft Real-Time Performance Through Better Slack Reclaiming,” in *Proceedings of the 26th IEEE Real-Time Systems Symposium*, 2005, pp. 410–421.

[34] G. Taylor, P. Davies, and M. Farmwald, “The TLB Slice–A Low-cost High-speed Address Translation Mechanism,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 355–363.

[35] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, “Compiler-directed Page Coloring for Multiprocessors,” in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 244–255.

[36] T. Sherwood, B. Calder, and J. Emer, “Reducing Cache Misses using Hardware and Software Page Placement,” in *Proceedings of the 13th International Conference on Supercomputing*, 1999, pp. 155–164.

[37] S. Cho and L. Jin, “Managing Distributed, Shared L2 Caches Through OS-level Page Allocation,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 455–468.

[38] D. Tam, R. Azimi, L. Soares, and M. Stumm, “Managing Shared L2 Caches on Multicore Systems in Software,” in *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.

[39] X. Ding, K. Wang, and X. Zhang, “SRM-Buffer: an OS Buffer Management Technique to Prevent Last Level Cache from Thrashing in Multicores,” in *Proceedings of the 6th ACM European Conference on Computer Systems*, 2011, pp. 243–256.

[40] N. Kim, B. C. Ward, M. Chisholm, C.-Y. Fu, J. H. Anderson, and F. D. Smith, “Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning,” in *Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS ’16, 2016.

[41] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, “Making Shared Caches More Predictable on Multicore Platforms,” in *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2013.

[42] J. Calandrino and J. Anderson, “Cache-Aware Real-Time Scheduling on Multicore Platforms: Heuristics and a Case Study,” in *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, July 2008.

[43] H. Kim, A. Kandhalu, and R. Rajkumar, “A Coordinated Approach for Practical OS-level Cache Management in Multi-core Real-Time Systems,” in *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS)*, July 2013.

[44] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, “Real-Time Cache Management Framework for Multi-Core Architectures,” in *Proceedings of the 19th IEEE International Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013.

[45] F. Bellosa, “Process Cruise Control: Throttling Memory Access in a Soft Real-Time Environment,” University of Erlangen, Germany, Tech. Rep. TR-14-97-02, July 1997.

[46] R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjdin, “The Multi-Resource Server for Predictable Execution on Multi-core Platforms,” in *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2014, pp. 1–12.

[47] L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale, and R. Bradford, “Single Core Equivalent Virtual Machines for Hard Real-Time Computing on Multicore Processors,” University of Illinois at Urbana-Champaign, Tech. Rep., October 2014.

[48] H. Zhu and M. Erez, “Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems,” in *Proceedings of the 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.