DNA: Dynamic Resource Allocation for Soft Real-Time Multicore Systems

Robert Gifford Neeraj Gandhi Linh Thi Xuan Phan Andreas Haeberlen
University of Pennsylvania
Email: {rgif, ngandhi3, linhphan, ahae}@cis.upenn.edu

***Abstract*—Modern latency-sensitive and real-time systems often use multi-core platforms; thus, tasks on different cores share certain hardware resources, such as the memory bus and certain cache levels. This has two undesirable consequences: (1) tasks can interfere with each other, causing high latency for the system as a whole, and (2) it becomes difficult to meet deadlines, since the worst-case timing of a given task depends on all the tasks it might have to compete with. Static partitioning isolates tasks from each other by allocating a certain fraction of the resources to each; however, many tasks execute in different *phases* (e.g., memory-intensive and CPU-intensive) that have different requirements. Thus, system designers are left with a choice between overprovisioning, based on the most demanding phase, or suboptimal performance.**

In this paper, we propose a pair of techniques, called DNA and DADNA, to address the above challenge. DNA increases throughput and decreases latency, by building an execution profile of each task to identify the phases, and then dynamically allocating resources based on which task can benefit the most; DADNA further adds support for soft real-time workloads by taking deadlines into account. We have built a prototype of both techniques in the Xen hypervisor; our experimental results show that, compared to a state-of-the-art solution, DNA and DADNA can substantially improve schedulability, reduce job deadline miss ratios, and cut latencies by more than a factor of two even in extremely overloaded situations.

I. Introduction

Today, multiple cores are a common feature of both desktop and embedded CPUs, and latency-sensitive and real-time systems are taking advantage of them to support their increasingly complex workloads. This is mostly a good thing, since having multiple cores means better performance. However, this trend also creates some new challenges because the cores are not independent – they share certain hardware resources, including the memory bus and certain cache levels. Thus, tasks on different cores can influence each other's runtime.

Consider, for instance, a system whose workload includes the following two kinds of tasks: (1) a control task, which multiplies a vector with a large matrix and then uses the result to make a complicated control decision, and (2) a stream-processing task, which computes a filter and then applies the filter to a stream of sensor inputs. (Both matrix multiplication and filter are basic functions in many real-time control systems.) Matrix multiplication is memory-intensive; on a single-core CPU, it can be fast if the matrix is already in the cache, or fairly slow if the matrix needs to be fetched from main memory. But on a multi-core CPU, it can be *even slower* if other cores happen to generate heavy traffic on the memory bus at the time the matrix needs to be fetched – for instance, if the stream-processing task happens to read the data stream at the exact same time. If this kind of interference happens unexpectedly, it can cause increased latencies and deadline misses.

One way to do better is to statically partition the resources (memory bandwidth and cache capacity) between the cores. For instance, MemGuard [77] regulates the memory bandwidth each core can use by counting memory accesses using hardware performance counters, and by interrupting workloads when a specific limit is reached; similarly, Intel's Cache Allocation Technology (CAT) [28] can restrict cores to a certain subset of the available cache partitions. These techniques can be used to prevent interference and to isolate the cores from each other. In our example, if the matrix-multiplication task and the stream-processing task run on different cores, we can allocate most of the last-level cache to the control task, since matrix multiplication benefits from caching but stream processing does not (and would in fact thrash the working set of the control task), and we can split the memory bandwidth between the two tasks, to make their execution times more predictable. It is well known that different kinds of tasks can extract different benefits from resources (see, e.g., the animalistic taxonomy from [68]), and this approach has been used for static partitioning [69].

However, static partitioning is *still* very conservative and often fails to fully utilize the available resources. The reason is its assumption that neither the set of tasks nor the characteristics of these tasks can change over time. In practice, the set of tasks often does change: for instance, the controller on a car might launch a new task when the driver enables cruise control, and stop it again when the driver switches back to manual mode.

More importantly, the characteristics of the tasks themselves can change as well! To see why, consider again our earlier example. During the matrix multiplication, the control task heavily depends on memory bandwidth, and we can speed it up considerably by allocating more bandwidth to it. However, computing the actual control decision afterwards is much less memory-intensive; during this time, it would be better to allocate most of the bandwidth to the stream-processing task. In other words, tasks can have different *phases* during their execution [56], and these phases can differ in their resource demands. Because of this, a static allocation – perhaps based on the demands of the worst phase, or on the average demand across phases – almost inevitably leads to suboptimal utilization and/or higher response times.

In this paper, we present a pair of resource allocation

techniques, Dynamic Allocation (DNA) and deadline-aware DNA (DADNA), that allocate memory bandwidth and cache capacity while explicitly taking the phases into account. Both techniques consist of two parts. The first is an offline profiler that runs each task with different combinations of resources to build an *execution profile* – essentially a function that maps different points in the task to the *rate of progress* (i.e., instructions retired per unit time) at that point. For instance, the profiler might find that the control task's matrix multiplication runs at 10^9 instructions/s if it can have the entire cache and the entire memory bus, but only at 10^8 instructions/s if it is restricted to 10% of the memory bus, and only at 10^7 instructions/s if it is additionally restricted to 50% of the cache (because of thrashing).

As a next step, the profiler then uses a simple machine-learning technique (clustering) to identify phases with similar behavior. This saves space – since we only need to store the phases and not the entire, detailed execution profile – and, more importantly, it identifies potential decision points at which it may make sense to adjust the resource allocation at runtime. The sequence of phases depends not only on the program but also on the resources the task has been allocated: for instance, memory-intensive phases can disappear when the tasks are given more cache space, or they can move around when cache partitions of different sizes cause different conflict patterns.

The second part of DNA/DADNA is a resource allocator that uses the collected execution profiles to dynamically reassign resources at runtime. Somewhat analogous to Antfarm [46], which dynamically allocates network bandwidth to BitTorrent swarms, our allocator gives memory bandwidth and cache capacity to the tasks that can benefit the most. Thus, in the above example, the control task would get most of the cache, since it benefits heavily from caching but the stream-processing task does not, and it would initially get most of the memory bandwidth, but only until the matrix has been loaded into the cache; after that, much of the bandwidth would be reallocated to the stream-processing task. The deadline-aware variant (DADNA) is additionally able to allocate resources not only based on the immediate needs of a task but also based on the slack time of current and future deadlines.

We have built a prototype implementation of DNA and DADNA in the Xen hypervisor [3] by modifying Xen's existing Real-Time Deferrable Server (RTDS) scheduler [50], and we report results from an experimental evaluation on real hardware. Our results show that DNA and DADNA incur only small run-time overhead, and that they can substantially improve schedulability, reduce deadline miss ratios, and cut latencies by more than a factor of two compared to a state-of-the-art solution. In summary, this paper makes the following contributions:

- the concept of phase-aware allocation (Section III);
- a phase-based task model (Section IV);
- the DNA resource allocation technique (Section V);
- DADNA, a deadline-aware variant of DNA (Section VI);
- a prototype implementation in Xen (Section VII); and
- an experimental evaluation (Section VIII).

II. Related Work

Sharing-aware analysis: One way to achieve timing guarantees in the presence of shared resources is to factor the sharing-related overhead into the timing analysis, as is done for memory, e.g., in [52], [51], [53], [30], [75], [15], and for caches, e.g., in [67]. However, without isolation, it is difficult to obtain tight bounds because one generally has no choice but to assume worst-case interference from other tasks or cores, which leads to a high latency overhead.

Resource partitioning: Another approach is to explicitly divide up the shared resources among the cores or tasks, and to strictly enforce this allocation at runtime. For memory bandwidth, hardware-based techniques, such as [79], [24], [25], [22], [36], can provide fine-grained control, but they are not available in most commodity processors; software-based solutions, such as [77], [78], [1], [76], typically leverage existing hardware features, such as the processor's performance monitoring unit. On the cache side, software-only approaches – such as page coloring [31], [40], [73] or compiler-based [43] techniques – are more limited, but fortunately, modern processors increasingly have explicit support for cache partitioning, e.g., in the form of Intel's CAT [28] or the Lockdown-by-Master (LbM) technology in ARM processors [2]. These techniques enforce a given allocation but cannot decide *how* to best allocate the resources among the tasks, which is the focus of the present paper. However, we rely on two of them – MemGuard [77] and Intel's CAT – to enforce DNA's and DADNA's allocations (see Section III-A for additional details).

Multiple resources: The solution we propose is able to 1) support latency-sensitive and soft real-time workloads, 2) take into account the intertwined relationship among three different resources (CPU, cache space, and memory bandwidth), and 3) consider the dynamic behavior of the workload, in the form of phases. There are several prior systems that can provide some subset of these properties, but, to our knowledge, there is none that can provide all three. Several systems are able to allocate more than one resource at the same time; for instance, [38] allocates cache space and memory banks; DRF [21] allocates CPU and memory; Quasar [17] allocates nodes, cores, memory, and storage; and [6], [65], [35], [54], [61], [44] allocate cache space and memory bandwidth. However, these systems focus on throughput (or, in the case of CoPart [44] and DRF [21], on fairness) and do not consider worst-case latencies or deadlines. Other systems do consider timing constraints but their resource allocations only take into account various combinations of *two* resources – such as CPU and memory bandwidth [72], [1], [45], [41], or cache and memory banks [63], [33], [32], [11] – but not all three. We are aware of only two systems, MARACAS [74] and CaM/vC^{2}M [70], [69], that can both support real-time workloads and consider all three resources, but both systems use static allocations and do not change the allocation based on dynamic behavior (i.e., the execution phases).

Prior work has considered dynamic allocation, but focuses primarily on individual resources. For instance, [62] changes

allocations based on marginal gain, but a) it focuses on HPC workloads without deadlines, and b) it measures the marginal gain at runtime, using counters, which works fine for one resource but would be difficult for, e.g., both memory bandwidth and cache. vCAT[71] introduces an abstraction for virtualizing Intel's CAT and a way to control it at runtime, but requires the programmer to manually insert system calls at phase boundaries to adjust the partitions; in contrast, our solution finds the phases without developer input, and it uses DNA/DADNA to make allocation decisions automatically.

Program types and phases: The insight that programs can have different interactions with resources is not new; previous work has classified the behavior using different colors [37], based on marginal utility [47], using miss models [9], via analytical modeling [4], or with animal types [68]. Zhuravlev et al. [80] compared some of these schemes and designed a (non-real-time) scheduling algorithm that uses them. Other work has used dependencies between resource usage and program inputs for scheduling, e.g., to save energy [20], [26]. The observation that the same program can go through different *phases* goes back to a paper by Sherwood and Calder [56], and since then, a number of techniques for identifying the phases have been developed, including ones based on k-means clustering [58], threshold clustering [18], [19], [59], visual inspection [14], pattern matching [34], [55], or wavelets [12], [13], [27], [55]. DNA and DADNA could leverage these techniques instead of the approach we used in our prototype (see Section IV). Phase-based workload characterization has also been exploited to build accurate energy models [23] and execution time prediction [49]. To our knowledge, however, DADNA is the first algorithm to use phases for the multicore resource allocation of latency-sensitive and real-time workloads.

III. Overview

We assume that the system consists of a set of tasks that execute on a shared multi-core platform, with a shared cache and a shared memory bus that are accessible by all cores. As in existing work, the set of tasks that can *potentially* run on the system is known before the system is launched. This is necessary because our approach involves some offline profiling to identify the phases and their resource requirements (Section IV). However, the set of tasks that are *actually* running on the system can change over time (e.g., in multi-mode systems [10]). We assume that the tasks are short and typically perform similar operations on similar inputs, which is true in most real-time systems where each task periodically executes a control function on streams of sensor inputs. This assumption allows us to use a relatively simple method to identify the phases (Section IV) but is not fundamental; if the tasks are more complex, there are other ways to find phases (e.g., [57]) that can be used instead.

We focus on latency-sensitive and soft real-time systems (though it should be possible to extend our solution to hard real-time systems). For soft real-time systems, we assume that deadline misses are acceptable, and jobs that miss their deadlines are allowed to continue their executions until completion. Our goals are 1) to reduce average, tail, and worst-case latencies, and 2) to minimize job deadline miss ratios when tasks have (soft) deadlines.

A. Background: CAT and MemGuard

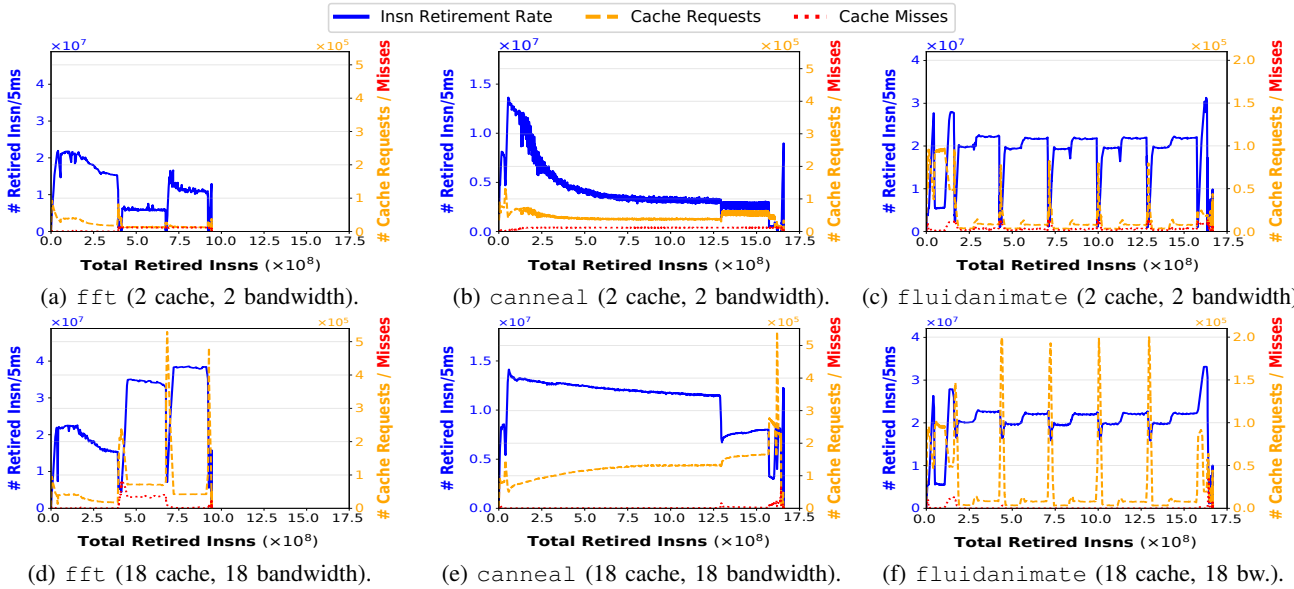
Intel's Cache Allocation Technology [28] is a hardware feature that is present in newer Intel CPUs; it allows the hypervisor to control how the shared last-level cache (LLC) is allocated between the physical cores. CAT divides the shared cache into N equal-size partitions (e.g., $N = 20$ on our evaluation machine), which can be allocated to one of several classes of service (COS). For each COS, there is a model-specific register with a bitmask – the capacity bitmask, or CBM – that controls which partition(s) should be used; for each logical core, there is another register (PQR) that specifies the COS for this core. CAT enforces the property that all new cache allocations from a logical core are made only in cache partitions specified in the CBM of that core's COS. For instance, if we set bits 0..3 in the bitmask for COS 1, and then set the PQR register for core #5 to 1, new cache allocations from core #5 will be made in one of the first four cache partitions.

MemGuard [77] is a software-based mechanism for enforcing per-core limits on memory bandwidth consumption. It uses the CPU's performance counters to count the LLC misses on each core; since memory bandwidth is consumed in response to LLC misses, this number is a good proxy for memory bandwidth consumption. Each core and/or each task can be assigned a memory bandwidth budget, and MemGuard periodically configures each core's counters so that they will generate an interrupt if the core's LLC misses exceed the budget for that period; if this interrupt is received, it throttles the core, or switches to a different task. In this paper, we use MemGuard to assign memory bandwidth in small, discrete partitions, just like CAT. (We could enforce limits at the granularity of a single LLC miss, but small differences in the bandwidth usually do not cause big differences in performance, and the small number of partitions helps us keep the number of configurations small.)

B. Case study: Setup

Since our approach is based on the fact that tasks often go through different phases with different resource requirements, we ran an experiment to illustrate this, and to show which phases exist in a typical workload. Specifically, the workloads we examined were the PARSEC [5] and SPLASH2x [66], [60] benchmark suites, which have often been used as workloads by prior work in this area [31], [30], [71], [64], [69], [70].

To get a platform where we could vary both the cache and memory bandwidth allocations, we used a Xen modification from our earlier work [69] that already supports MemGuard. We extended it to additionally support Intel's CAT. We ran the modified Xen on a CAT-capable Intel Xeon E5-2683 v4 processor with 16 cores and a 40MB 20-way set-associative L3 cache that is shared among the cores. (Each core has its own L1 and L2 caches.) This processor has 16 COS registers and supports 20 L3 cache partitions. The machine also had three single-channel 16GB PC-2400 DDR4 DRAMs. Using the



(a) `fft` (2 cache, 2 bandwidth). (b) `canneal` (2 cache, 2 bandwidth). (c) `fluidanimate` (2 cache, 2 bandwidth).

(d) `fft` (18 cache, 18 bandwidth). (e) `canneal` (18 cache, 18 bandwidth). (f) `fluidanimate` (18 cache, 18 bw.).

Fig. 1: Execution patterns for three benchmark tasks under two different resource allocations: two cache and two bandwidth partitions (top row), and 18 cache and 18 bandwidth partitions (bottom row). Each point on the horizontal axes represents a particular point in a program, identified by the number of instructions since that program was started, and the lines show the behavior of the program at that point: its execution speed (blue) and the rate of cache requests (orange) and cache misses (red).

method in [78], we measured a maximum guaranteed bandwidth of 1.4 GB/s, which we divided into 20 partitions of 70MB/s each. This is lower than the peak bandwidth that the platform supports, but it results in better isolation between the cores. To avoid nondeterministic timing, we disabled hyperthreading, SpeedStep, and hardware prefetching.

To collect data for a given benchmark task and a given cache/bandwidth allocation, we booted the Xen hypervisor with its built-in RTDS real-time scheduler [50] and then launched one guest VM running LITMUS^{RT} [7] (a real-time OS) with two VCPUs that were each pinned to a dedicated core; one to run essential guest OS tasks and the other to run our benchmark in isolation. We then ran the benchmark task on this VCPU, synchronized its release with the VCPU's release (as in [69]), and we measured three performance metrics, using the CPU's performance counters: (i) the total number of cache requests, including both hits and misses; (ii) the number of cache misses, as an indication of the traffic on the memory bus; and (iii) the number of retired instructions. For each configuration of cache and memory bandwidth resources, we took a set of measurements every Δ milliseconds and collected the sets of measurements for 100 runs. (We used $\Delta = 5$ms, as it was small enough to capture fine-grained changes in resource use patterns for our workloads, without creating too much noise.)

C. Case study: Results

Figure 1 shows our results for three representative benchmark tasks – the `fft` program from SPLASH2x, which performs signal processing, and the `canneal` and `fluidanimate` programs from PARSEC, which perform simulated annealing and fluid dynamics for animation purposes, respectively – as well as for two different resource allocations: one in which resources are scarce and the task is given only two (10%) of the 20 available cache and memory bandwidth partitions (top row), and another in which resources are plentiful and the task is given 18 (90%) of the 20 partitions (bottom row). Each graph shows three curves: the blue one shows the number of instructions retired in the preceding 5ms window, the orange one is the number of cache requests, and the red one is the number of cache misses, which corresponds to memory traffic. The horizontal axes identify particular points in each program: for instance, the `fft` program executes about $9{\cdot}10^8$ instructions, so the lines in both Figures 1(a) and (d) end at that point; the speed (blue line) is roughly comparable during the first $4 \cdot 10^8$ instructions, but the rest of the program runs much faster when more cache and bandwidth partitions are available.

A look at the top row of Figure 1 shows evidence that different execution phases do exist: for instance, `fft` has three clearly separated phases, of which the first shows quick progress and almost no cache misses (due to high locality), while the other two show slower progress (but at different rates!) and very high miss rates. `fluidanimate` shows a cyclic behavior with three alternating phases that vary substantially in the number of cache accesses, and `canneal` has one long phase, followed by a much shorter one. This leads to our first finding:

Finding 1. *A task's resource usage patterns vary throughout its execution and can be broadly divided into phases, where each phase exhibits a distinct cache and memory bandwidth resource demands.*

Our next observation is that, while each of the three tasks does exhibit different phases, both the number and the characteristics of the phases are quite different: in `fft`, there are three large phases, with rates of execution changing little within

a phase but substantially between phases; `fluidanimate` has a sequence of shorter phases, including some brief spikes; `canneal` has some variation within its long first phase; etc. This leads to our second finding:

Finding 2. *The number of phases and resource demand patterns in each phase are different across different programs.*

A third observation is that spikes in the orange and red curves (cache and memory bandwidth) generally correspond to dips in the blue curve (progress). This should not be surprising: the more often a task needs to access the cache and/or main memory in a given phase, the slower it will be. Conversely, we can *generally* expect to speed up a task by giving it more space in the cache or more memory bandwidth. (There are exceptions – e.g., when a task's working set already fits into the allocated cache space, as well as some unusual cases we discuss below.) We summarize this in our third finding:

Finding 3. *The execution rate in each phase is closely related to the resource demands in that phase: more cache misses or cache requests tend to lower the execution rate, and fewer requests tend to increase it.*

So far, we have focused mostly on the top row of Figure 1. We now compare the top row (scarce resources) to the bottom row (plentiful resources); notice that the horizontal axis is the number of instructions retired, so the same horizontal point in the two rows corresponds to the same point in the execution. Notice how *some* phases change their characteristics substantially (e.g., the third phase of `fft`, which now has no more cache misses and runs much faster), while others change little, if at all (e.g., the phases of `fluidanimate`, which have fewer cache misses but run at pretty much the same speed). This is expected: once the resource demands of a phase are satisfied, increasing the allocation further should not have much of an impact on the rate of execution.

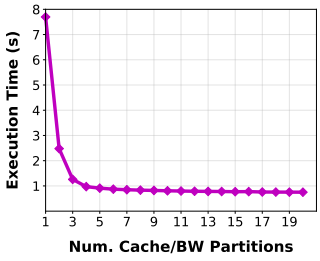


Fig. 2: Runtime for `canneal` for different allocations.

To reinforce this point, Figure 2 shows how the overall execution time of `canneal` changes as we vary the number of cache and memory-bandwidth partitions it can use. (For simplicity, we use the same number of partitions for both.) Once the number of partitions reaches a certain threshold (around 5), further allocations do not change the execution time very much. This leads to our final observation:

Finding 4. *Allocating extra cache and memory bandwidth resources to a task can help improve its execution time, but only up to a certain number of partitions.*

In this section, we have focused on the typical behavior we have seen in our experiments. However, we have also noticed a number of atypical events. For instance, new phases can appear, or existing ones disappear, as the resource allocation is changed – e.g., because a larger cache allocation can, in combination with certain cache replacement strategies, cause a form of thrashing. Also, while Figure 2 shows "smooth" changes in runtime as the allocations are changed, this behavior is not universal; sometimes there is a threshold effect where thrashing persists until a certain cache allocation is reached, but then disappears abruptly. This will become important in Section V.

IV. DNA: Phase generation

Next, we describe how, based on the findings from the previous section, we build a model that captures the phases of a given task, along with their resource requirements.

A. What is a phase?

Since the resource allocation algorithm will need to make decisions based on a task's current phase, we need a way to quickly tell, from the "outside", which phase a given task is currently in. The instruction pointer is not a good option for this – partly because of loops, but also because the same function can be invoked from different contexts. For instance, a `matrixMultiply` function could be compute-bound when invoked with a small matrix, and memory-bound when invoked with a large one.

Because of this, we use the *number of retired instructions* to estimate what part of the program is executing, and we define a phase to be a range of instructions – for instance, a phase could last from the 10,000th instruction to the 15,000th one. The number of retired instructions can be easily measured using the performance counter on Intel CPUs, and many other CPUs have a similar counter.

This definition is not precise: for instance, the count could be widely off if a task does busy waiting, if it is invoked with inputs of different sizes, or if the control flow varies widely depending on the inputs. However, for real-time systems, this approach is plausible because they often involve periodic tasks that perform the same operations again and again, on similar inputs. (For instance, the operations might be reading data from a particular sensor, filtering data, or making a control decision.) Small variations in the control flow are not problematic because we do not need to change the resource allocation precisely at a particular instruction, but rather when the task is entering a certain (long) phase, so all we really need is an approximate point.

For more complex workloads, this simple approach would not work, but there are other, more sophisticated techniques in the literature (e.g., basic-block vectors [57]) that could be used instead. (Precisely how the phases are delimited is somewhat orthogonal to our work; our main focus is resource allocation.)

B. Step #1: Profiling

At a high level, the DNA algorithm aims to allocate resources to the task(s) that will "benefit the most" from them – that is, the tasks whose rate of execution will increase the most. To do this, it needs to know, for a given task and a given instruction count within that task, what the rate of execution would be if

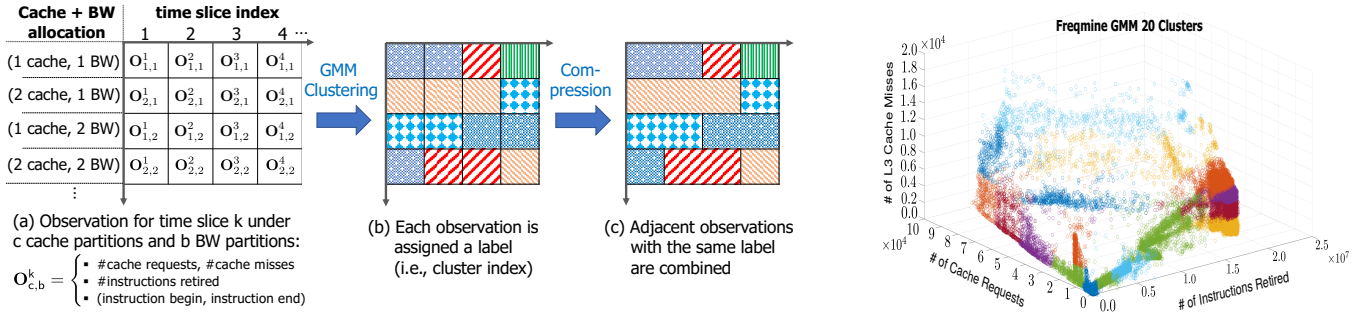


Fig. 3: Phase generation process for a single task (left) and clusters generated for the `freqmine` task (right).

the task were allocated a certain set of resources. We obtain this information through profiling.

The profiling process is the same as the one from our case study (Section III-B). The first step is to set up a carefully controlled environment in which 1) tasks can be run without interfering with other tasks, and in which 2) we can control the resources that each task has access to. In our experiments, we profile the tasks one by one, on a dedicated CPU core that no other task may access, and we disable all other workloads and all nonessential OS features, to prevent resource consumption by other, unrelated tasks; we also use CAT and MemGuard to control the number of cache partitions and the memory bandwidth the task has available to it.

Next, we run each task in this environment for $N = 100$ runs per resource allocation configuration (there are 400 configurations in total). In each run, for every $\Delta = 5$ milliseconds, we collect (1) the number of instructions completed; (2) the number of L3 cache requests (hits + misses); and (3) the number of L3 cache misses. This information can be easily gathered from the hardware performance counters on Intel CPUs, and similar counters are present on many other processors.

Finally, we replace each measurement with the delta over the measurement before it – that is, the number of instructions completed, cache requests made, and cache misses *since the last measurement*. We refer to each set of deltas as an *observation*; intuitively, an observation describes the activity of a task during a given Δ window. Together, the observations from the different runs form a matrix that is illustrated in Figure 3(a).

C. Step #2: Clustering

The next step is to automatically identify groups of observations that show similar behavior. This step could be done using a variety of clustering techniques; for our prototype, we used expectation-maximization (EM) [42], in combination with a Gaussian Mixture Model (GMM) [48], on a three-dimensional feature space (since each observation contains three metrics). A GMM model simply contains k Gaussian distributions, and the EM technique discovers a mean and a covariance for each that are a good fit for the specific data set. The process is somewhat similar to the classical k-means clustering: EM produces, for each observation, a posterior probability that the observation belongs to each of the k Gaussians, then it updates the mean and covariance for each Gaussian to the maximum-likelihood values for the associated observations, and it repeats these steps until convergence.

As is usually the case with clustering, the correct value of k is not known a priori, but we can generate clusterings with a range of different values for k ($2 \leq k \leq 30$ in our experiments) and then evaluate their output quality using the Davies-Bouldin index [16] and the Calinski-Harabasz index [8]. (The two metrics differ in how they weigh cluster density and separation.) We use the clustering with the highest quality; if multiple clusterings have the same quality, we use the one with the largest k (i.e., the most phases) among those to provide more fine-grained knowledge of the resource needs of the task.

As an illustration, the right side of Figure 3 shows the clustering for the `freqmine` task, with different colors representing different clusters.

D. Step #3: Identifying phases

The final step is to identify the phases. Recall that we need the phases to serve two purposes: they are a more compact representation of the (very verbose) profiling data, and they identify possible decision points for DNA and DADNA, where it "makes sense" to potentially change the resource allocation.

Finding phases in the original profiling data would be difficult because the observations usually are all different, and it is not clear which differences are significant. However, once we replace each observation with the label of the cluster it belongs to, we typically find long sequences of identical labels. (This is illustrated in Figure 3(b); recall that each row represents a run with a different set of resources.) We can then simply collapse each contiguous sequence into a single phase, keeping track only of 1) the instruction where the phase began, 2) the instruction where the phase ended, and 3) the average rate of execution (instructions executed per unit time) during the phase. The result is illustrated in Figure 3(c). This is the information the DNA and DADNA algorithms need.

V. DNA: Resource Allocation

In this section, we describe how DNA performs resource allocation. DNA is agnostic to deadlines and merely optimizes for overall system throughput and latencies. A deadline-aware version, DADNA, is presented in the next section.

Algorithm 1 The DNA algorithm

```
 1: function AllocateCachePartitions(τ, δc)
 2:     c[τ] = c[τ] + δc
 3:     rem_c = rem_c − δc
 4:
 5: function AllocateMemoryBandwidth(τ, δb)
 6:     b[τ] = b[τ] + δb
 7:     rem_b = rem_b − δb
 8:
 9: function GiveResourceTo(τ)
10:     c_gain = θ(τ, i(τ), c[τ], b[τ], rem_c, 0)
11:     b_gain = θ(τ, i(τ), c[τ], b[τ], 0, rem_b)
12:     if b_gain < c_gain then
13:         AllocateCachePartition(τ,1)
14:     else
15:         AllocateMemoryBandwidth(τ,1)
16:
17: function DNA(T, i)      ▷ T: running tasks, i: instr. completed
18:     rem_c = C                                  ▷ Max cache partitions
19:     rem_b = BW                       ▷ Max mem bandwidth partitions
20:
21:     /* Assign initial resources */
22:     for τ ∈ T do
23:         c[τ] = b[τ] = 0
24:         AllocateCachePartitions(τ, min_c)
25:         AllocateMemoryBandwidth(τ, min_b)
26:
27:     /* Iteratively refine allocations */
28:     while rem_c > 0 || rem_b > 0 do
29:         τ = argmax_τ{θ(τ,i(τ),c[τ],b[τ],rem_c,rem_b)}
30:         GiveResourcesTo(τ)
31:     return (c, b)          ▷ c,b map cache/bw partitions to tasks
```

A. Invocation and output

The purpose of DNA is to find an allocation of (cache and memory bandwidth) resources to cores, so as to maximize the total rate of execution for the entire system. DNA itself does not perform scheduling; it is designed to be used in combination with an existing scheduler. At the point DNA is invoked, the scheduler has already picked a task for each core to run, and DNA allocates resources to these running tasks. In our prototype, the scheduler is partitioned Earliest Deadline First (EDF), but other schedulers can be used as well. In principle, a single, more complex algorithm could make both decisions simultaneously; we do not consider this approach here, but it could be an interesting direction for future work.

DNA is deterministic, that is, given the same mapping of tasks to cores and the same parameters about the current phases, it will output the same resource allocation. Thus, it generally makes no sense to invoke it again unless there is a change in one of the two. In other words, DNA should run if either (a) the scheduler has changed the task on at least one core, or (b) one of the running tasks encounters a phase transition for *any* resource allocation.

The last point is a bit subtle; it is related to the observation from Section III-C that the same task can go through different phases if given different resource allocations. For instance, suppose a task has just finished a (memory-bound) matrix multiplication and now begins a (compute-bound) cryptographic signature, and consider the scenarios where the task is either given a large number of cache partitions L or a small number S. If the task had L partitions before, it might not have generated any memory traffic during the multiplication, and its overall rate of execution might not change much at the transition point; however, with only S partitions, its rate of execution would have been low before and would be high now. Because of this, DNA should run again at the transition point *even if* the task currently has L partitions and is still in the middle of its current phase – simply because there is another allocation (S) that would exhibit a phase transition, and because it may now make sense to switch to S partitions and allocate the remaining L-S partitions to another task.

DNA outputs a mapping of cache and memory bandwidth partitions to tasks. This mapping can then be enforced by the OS or hypervisor, e.g., with CAT and MemGuard.

B. Algorithm

In principle, a good resource allocation could be found using a form of multidimensional bin packing. However, this kind of computation is expensive and would generate a high overhead, especially since, as discussed above, DNA needs to run frequently. Because of this, we instead opt for a heuristic that can be evaluated quickly.

Algorithm 1 shows the algorithm for DNA. As a first approximation, DNA is a greedy heuristic: it starts by giving only the minimal allocation to each running task (lines 21–25) and then iteratively assigns an additional cache or bandwidth partition to the running task that can "benefit the most" – in other words, the task whose rate of execution would increase the most on average, relative to the allocation it has so far (lines 27–30). Intuitively, the function $\theta(\tau, i(\tau), c, b, \delta c, \delta b)$ is the "gradient" in the rate of execution of a task τ, after executing i instructions, when adding δc cache allocations and δb bandwidth allocations. θ can be thought of as the *sensitivity* of τ to a change in its resource allocation. θ can be computed from the data that is gathered during profiling.

However, this simple approach would not work very well by itself. The reason is that some tasks benefit very little from extra resources, unless and until they reach an allocation of a certain size (say, enough cache partitions to fit their entire working set) but at that point the benefit could be very large. If DNA made decisions based on only the *local* gradient – that is, the benefit from getting *one* extra cache or memory bandwidth allocation – it might never be able to reach the large benefit, since it would always seem that allocating one extra resource makes little difference.

To avoid this, we use a slightly different definition of θ that takes larger increases into account as well. Let $\rho(\tau, i, c, b)$ be the rate of execution of τ after i instructions, using c cache partitions and b bandwidth partitions. Then we define θ as:

$$\theta(\tau, i, c, b, \delta c, \delta b) := \frac{1}{\delta c \cdot \delta b} \sum_{j=0}^{\delta c} \sum_{k=0}^{\delta b} \rho(\tau, i, c+j, b+k) - \rho(\tau, i, c, b)$$

where δc and δb represent the amount of remaining available resources on the system to be assigned.

In other words, θ represents the *average* increase in the rate of execution when adding *up to* δc cache partitions and *up to* δb bandwidth partitions. This is the function used in line 29 of Algorithm 1. After a resource has been assigned, δc or δb will decrement by 1 and thus the task's sensitivity to more resources will update to be realistic with the amount of resources it can still receive. Notice that, in practice, we do not explicitly record θ; instead, θ can be computed efficiently from the (compact) phase information we derived in Section IV.

There is one final complication, which has to do with the fact that Algorithm 1 allocates the resources one by one, rather than in larger increments. Once DNA has picked, in line 29, a task to give additional resources, it must still decide *which* resource (cache partition or memory bandwidth partition) to allocate. We make this decision by comparing, in line 12, the *marginal* improvement each resource provides.

VI. Deadline-aware DNA

In this section, we present an extension of DNA, called DADNA, for soft real-time workloads that aims to minimize deadline misses (in addition to improving latencies).

A. Basic operation

Once tasks have deadlines, it is no longer enough to just allocate resources to the tasks that "benefit the most", in terms of rate of execution; we sometimes need to allocate extra resources to certain tasks just to enable them to finish before their deadlines. In other words, the optimization function becomes the total rate of execution, subject to the constraint that all tasks should meet their deadlines.

Algorithm 2 shows how DADNA achieves this goal. (Functions that were already defined in Algorithm 1 have been omitted for brevity.) The beginning is similar to DNA: in lines 13–20, we begin again by allocating the minimum resources to each task. However, we now also extrapolate, using a function called TimeLeft, for how much time each task τ would still need to finish, if given only this minimum allocation, and we compare this time to the slack $S(\tau)$ – that is, the amount of time τ has left before its next deadline. If the task cannot finish in time, it gets added to a set *prio* and will be prioritized during the rest of the algorithm. The only exception, in line 19, is for tasks that cannot finish in time at all, even if given all the available resources. These tasks would use up all the resources if they were added to *prio*.

Next, in lines 22–27, the DADNA algorithm allocates resources to the tasks in *prio*, starting at the task with the "greatest need" (that is, the greatest difference between its projected completion and its deadline). Once a task has enough resources to finish before the deadline, it is removed from the *prio* set. If there are resources left over when the set is empty, DADNA allocates them in the same way as DNA, by giving them to the tasks that can benefit the most (lines 29–32).

B. Virtual deadlines

So far, we have considered only the tasks that are *currently* running on the available cores. If the core scheduler is EDF,

Algorithm 2 The DADNA algorithm

```
1: function TimeLeft(τ, c, b)
2:     p = {x | P(τ,c,b).start ≤ i(τ) ≤ P(τ,c,b).end}
3:     left = (P(τ,c,b).end − i(τ))/P(τ,c,b).ρ
4:     for each j with p < j ≤ maxPeriod(τ,c,b) do
5:         left += (P(τ,c,b).end − P(τ,c,b).start)/P(τ,c,b).ρ
6:     return left
7:
8: function DADNA(T, i, S)                    ▷ S(τ): Slack of τ
9:     rem_c = C                               ▷ Max cache partitions
10:    rem_b = BW                              ▷ Max mem bandwidth partitions
11:    prio = ∅
12:
13:    /* Assign initial resources */
14:    for τ ∈ T do
15:        c[τ] = b[τ] = 0
16:        AllocateCachePartitions(τ, min_c)
17:        AllocateMemoryBandwidth(τ, min_b)
18:        if TimeLeft(τ,c[τ],b[τ]) > S(τ) then
19:            if TimeLeft(τ,rem_c,rem_b) ≤ S(τ) then
20:                prio = prio ∪ {τ}
21:
22:    /* Help tasks meet their deadlines */
23:    while (rem_c > 0 || rem_b > 0) ∧ (prio ≠ ∅) do
24:        τ = argmax_τ {TimeLeft(τ, c[τ], b[τ]) − S(τ)}
25:        GiveResourcesTo(τ)
26:        if TimeLeft(τ,c[τ],b[τ]) ≤ S(τ) then
27:            prio = prio \ {τ}
28:
29:    /* Iteratively refine allocations */
30:    while rem_c > 0 || rem_b > 0 do
31:        τ = argmax_τ {θ(τ,i(τ),c[τ],b[τ],rem_c,rem_b)}
32:        GiveResourcesTo(τ)
33:    return (c, b)                           ▷ c,b map cache/bw partitions to tasks
```

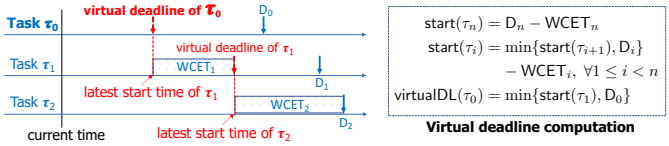


Fig. 4: Virtual deadline illustration and computation.

these will be the tasks whose deadlines are currently the closest. However, by allocating resources to these tasks based *only* on how much *they* need to finish in time, we are potentially harming other tasks that are in the ready queue and are not currently running.

The left picture of Figure 4 illustrates the problem. Here, task τ_0 has the earliest deadline and will currently be running. But if τ_0 is given just enough resources to finish by its deadline D_0, tasks τ_1 and τ_2, which have deadlines shortly thereafter, will be doomed: if they start to run at D_0, there is simply not enough time left to finish both by D_2, let alone D_1.

To fix this problem, we use a variant of an old trick: *virtual deadlines*. At a high level, this works as follows. We begin with the task in the ready queue that has the largest deadline (τ_2 in our example) and compute the latest point in time at which this task would need to be started in order to finish by its deadline, assuming it is given the maximum possible resource allocation. If the next-highest deadline (τ_1's, in our example) is after that

point, we replace it with a *virtual* deadline at that point. We then repeat this process with the earlier deadlines, until we arrive at a (possibly virtual) deadline for the currently running task τ_0. In other words, we (recursively) compute the virtual deadlines as shown in Figure 4, where τ_n denotes the task in the ready queue with the highest deadline, and D_i and WCET_i denote the absolute deadline and the worst-case execution time of τ_i under the maximum possible resource allocation. These virtual deadlines can then be used to compute the slack S for DADNA, as before. (By definition, the virtual deadline of the running task should be recomputed whenever a new job with a larger absolute deadline is released on the same core.)

Note that the virtual deadlines are a heuristic that boosts tasks that are urgent, and they are internal to DADNA only (i.e., the CPU scheduler never sees them).

Remarks: Like all existing multicore resource allocation algorithms (that we aware of), our algorithms are not optimal; there are cases where a schedule is theoretically possible, but DNA/DADNA will not find it. However, our experimental results suggest that DNA and DADNA work substantially better than the state-of-the-art technique in terms of schedulability, deadline miss ratios, and average/tail/worst-case latencies. Our experiments use DNA/DADNA with partitioned EDF, but DNA/DADNA should work with any CPU scheduler (though the benefits could vary).

In this work, we focus on reducing latencies and minimizing job deadline miss ratios; however, with a schedulability test, our solution can be adapted to hard real-time systems as well. Since DNA/DADNA is deterministic, one way to obtain a simple schedulability analysis for periodic tasks is to run DNA/DADNA for an entire hyperperiod, and to assume that, in each phase, each task runs for the maximum time we observed for that phase during profiling. A closed-form analysis would not be trivial, because the execution time depends on the allocated resources, but should still be possible.

VII. Prototype implementation

To evaluate our solution and to show that it can be integrated into a practical run-time system, we built a prototype of our solution on top of the Xen hypervisor. In this section, we describe some key aspects of this prototype.

Partitioning mechanisms: For partitioning the cache and the memory bandwidth, we built on top of a patch to Xen 4.8 from our earlier work [69]. This patch contains support for the MemGuard [77] technique, and we extended it to additionally support Intel's CAT. As discussed in Section III-B, we artificially partition MemGuard's (continuous) memory bandwidth limits into fixed-size "partitions", whose number is equal to the number of cache partitions.

Soft-real-time support: We further extended Xen's RTDS scheduler to enable multiple instances of a VCPU to co-exist in the run queue. This is necessary to support soft real-time systems, where jobs may execute beyond their deadlines.

Thread support: Our current prototype is restricted to single-threaded workloads. This is not inherent; the reason is simply that our phase characterization (Section IV-A), which is based on the number of retired instructions, works best if the programs are deterministic. However, we could use deterministic multi-threading – e.g., Dthreads [39] – to add thread support without losing this property (and with comparable performance), or we could use a different way to identify where a phase begins and where it ends.

Phase generation: Our prototype includes the phase generation technique from Section IV. To collect observations, we extended Xen's RTDS scheduler with a configurable timer, and we added a timer handler that recorded three CPU performance counters every $\Delta = 5$ ms. (Note that profiling is done one task at a time, so EDF scheduling is not necessary.) As discussed in Section IV-B, we set up the performance counters to track, on each core, (i) the number of instructions retired, (ii) the total number of L3 cache requests, and (iii) the number of L3 cache misses. To prevent interference from the hypervisor itself, we configured the performance counters to prevent counting at the hypervisor's privilege level.

DNA/DADNA with partitioned EDF: We implemented DNA/DADNA in our extended Xen's RTDS scheduler. The scheduler uses partitioned EDF scheduling, where tasks are restricted to a specific core, as it has smaller run-time overhead. We use worst-fit bin packing to assign tasks to cores (though other bin-packing algorithms can also be used). This has the effect of simplifying the virtual-deadline calculation for DADNA, since the number of tasks that can run on a given core and must be considered in this calculation is typically small. Overall, our implementation consists of approximately 960 lines of code for DNA and an additional 200 for the extension to consider deadlines in DADNA. For simplicity, our implementation of DADNA made one small simplification to Algorithm 2: instead of using the `TimeLeft` function, which estimates the remaining time based on the current *and* future phases, our code extrapolates based on just the current phase. This sometimes causes DADNA to make suboptimal decisions, so our results in Section VIII are slightly conservative.

Thrashing avoidance: To avoid cache thrashing, we set the minimum number of cache partitions a task can receive to $min\_c := 3$. We also take care to minimize the number of cache partitions that need to be reallocated when an allocation changes. This is not trivial because Intel's CAT requires each core to have a contiguous range of partitions [29, §17.19.2]: for instance, a core can get partitions #5–10, but not partitions #4–6 and #8–10. Fortunately, DNA gives us some flexibility because it only assigns each task a certain *number* of cache partitions, without specifying which ones. Thus, we can use the following simple heuristic to allocate contiguous ranges: core #0's range always starts at partition #0, the last core's range always ends at the last partition, and the ranges in between are ordered by core number. For instance, if there are four cores and DNA assigns (7,6,4,3) partitions to the tasks on these cores, the cores will get ranges #0–6, #7–12, #13–16, and #17–19. Thus, if DNA next assigns (6,7,4,3), we can simply reassign partition #6 from core #0 to core #1. Hypothetically, #14–19,

#0–6, #7–10, and #11–13 could also be used, but this would involve reassigning every single partition to a different core.

Decision points: For ease of implementation, our DNA/DADNA prototype made one simplification: instead of running precisely at phase boundaries of the scheduled task, we run DNA/DADNA (i) periodically at 1ms intervals and (ii) whenever a new task is scheduled onto a core. This choice adds a small performance penalty, since DNA/DADNA may run more often than strictly necessary, and since a task may need to wait for a few microseconds after a phase change before its allocation changes accordingly, but we do not expect these costs to be significant. When DNA returns an allocation that is different from the current one, we update MemGuard's bandwidth limits directly from within the hypervisor, and we use the `wrmsr` instruction to update the bitmasks in the COS registers with the new mapping of cache partitions to cores.

VIII. Experimental Evaluation

To evaluate our solution, we performed an experimental evaluation using our prototype. Our key questions were: (1) What is the run-time overhead of DNA/DADNA? And (2) Can DNA and DADNA indeed improve latency, job miss ratio and schedulability, compared to a state-of-the-art solution?

A. Experimental setup

Baseline: We compared DADNA and DNA to vC^{2}M [69], a state-of-the-art resource allocation technique that supports real-time workloads and can handle both cache partitions and memory bandwidth, but *cannot* take phases into account. vC^{2}M is an extension of [70], which has already been shown to substantially outperform systems without resource management, so we omit a "free-for-all" baseline in which the tasks compete for resources without any constraints. vC^{2}M takes the WCETs for each task in the workload as an input; once tasks have been assigned to cores, it *statically* assigns a number of cache and memory bandwidth partitions to each core, so as to maximize resource utilization while meeting the deadlines, but without considering in detail the behavior of the tasks. vC^{2}M comes with a schedulability analysis and is thus able to support both soft real-time and hard real-time workloads; here, we focus on the former, since DNA and DADNA support only soft real-time workloads.

Workload: Since resource allocation techniques need to work for a wide variety of workloads, it is customary to evaluate them with synthetic workloads, so that a large part of the design space can be covered. Following the approach from [69], we randomly pick our tasks from a widely used multithreaded benchmark suite; however, while [69] considered only PARSEC [5], we also use tasks from SPLASH2x [60], to get a somewhat larger variety. Both benchmark suites support a single-threaded execution mode, which we used. We profiled the tasks as described in Section III-B, using the `simsmall` input type, and we extracted the phase information as discussed in Section IV; the profiling step also yields the WCET for each resource allocation, which is required by vC^{2}M, as well as a *reference WCET*, which is the task's WCET when it is allocated the *entire* cache and the *entire* memory bus. We picked the tasks' utilizations uniformly at random from $[0.1, 0.4]$, and we set the period (deadline) of each task to be its reference WCET divided by its utilization.[1] We generated task sets with taskset utilizations ranging between 1.0 and 3.8, at steps of 0.2. Notice that these utilizations are calculated using reference WCETs as well, i.e., on the assumption that each task can have the entire cache and memory bus to itself, when in practice the tasks have to share. Because of this, a utilization of 2.6 on four cores is already heavy and a utilization of 3.0 fully overloads the system (as they would correspond to a utilization of 3.6 and 4.1, respectively, if we assumed the cache and memory bandwidth were divided evenly among the cores). For each taskset utilization, we generated 15 independent tasksets, for a total of 225 tasksets.

Platform: We ran the generated workloads on the machine described in Section III-B, using the exact same platform setup. Each task was pinned to its own dedicated VCPU, which in turn was pinned to one of four cores that was selected by worst-fit bin packing. A fifth core was reserved for running essential OS services.

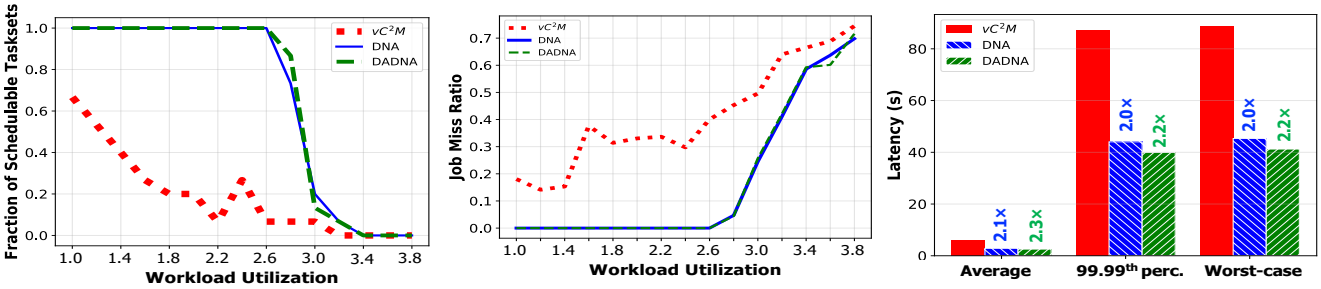
Experiments: For each taskset, we released jobs during a two-minute interval and ran them until completion under each of the three algorithm settings, and we collected the response times of all jobs (that is, the interval from the instant the job is released until the instant it is completed).

B. Run-time overhead

Both DNA and DADNA must run frequently, so they can respond quickly to phase changes and task additions or terminations. Thus, it is important that they can run quickly and do not add a significant overhead. In general, DNA's overhead depends on the number of cores, while DADNA's overhead depends on the number of tasks per core. To examine the cost, we performed the following experiment: we used Xen's time interface to measure the duration of each DNA or DADNA run; this includes both the time to run the algorithms themselves and the time to change the cache and/or bandwidth allocations.

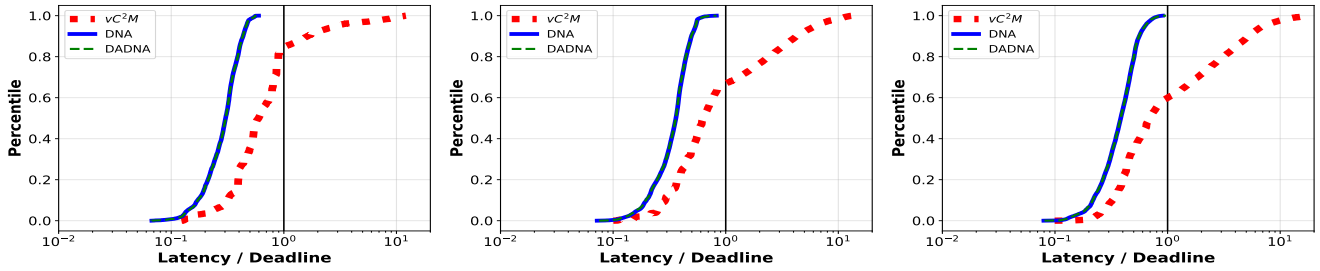
Table I shows our results. On average, both algorithms take about 16μs to run, so, if they are invoked every millisecond, the overhead is about 1.6%. (Notice that our implementation is unoptimized, and that the overhead could be further reduced by running DNA/DADNA only at phase change points, rather than periodically, as discussed in Section VII.) The 99^{th} percentile and the maximum are higher. This is because adjusting resource allocations is expensive: changes to both the COS registers and the performance counters involve writing a model-specific register, which can take thousands of cycles on our platform. Normally, there are few changes, so these costs add at most a few microseconds, but in rare cases, most or all of the allocations have to be changed, which results in a higher cost. Overall, the run-time overhead of DNA/DADNA is

[1] We also evaluated the algorithms using tasksets with bimodal utilization distributions, and the results were consistent with that of tasksets with uniform distributions. Due to space constraints, we omit the details.



(a) Schedulability. (b) Job miss ratio. (c) Latency.

Fig. 5: Performance of DNA, DADNA, and vC^{2}M across all workloads.



(a) Workload utilization 1.4. (b) Workload utilization 2.0. (c) Workload utilization 2.6.

Fig. 6: CDF of normalized latencies.

	Average	99^{th} Percentile	Maximum
DNA	16.07μs	46.11μs	103.44μs
DADNA	16.36μs	46.48μs	110.23μs

TABLE I: Run-time cost of DNA and DADNA.

reasonably small, and this overhead is already factored into their performance benefits reported in the following subsections.

C. Schedulability and deadline miss ratio

By dynamically giving resources to the tasks that can (currently) make the best use of them, DNA and DADNA should be able to improve the throughput and reduce latency, relative to a static allocation. Thus, the system should be able to schedule bigger workloads. Our first experiment tests that hypothesis. We ran experiments with DNA, DADNA, and vC^{2}M, using workloads with different utilizations, and we measured the fraction of tasksets that were empirically schedulable – i.e., tasksets whose jobs *all* meet their deadlines during our experiment.

Figure 5(a) shows our results. As expected, as the workload utilization increases, the fraction of schedulable tasksets also decreases for all algorithms. However, we observe that DNA and DADNA are able to schedule much larger workloads than vC^{2}M, due to their more effective use of the available resources. For instance, at a workload utilization of 2.6, vC^{2}M was able to schedule only 6.67% of the tasksets, whereas DNA and DADNA were able to schedule all tasksets (a 15$\times$ improvement). Hence, the latter is a clear improvement over the former in terms of schedulability.

Figure 5(b) shows how the job miss rate varies with the workload utilization. vC^{2}M experiences a substantial miss rate from very early on: even at a utilization of 1.6, it already incurs more than 30% miss rate. In contrast, DNA's and DADNA's miss rates remain zero for all utilizations up to 2.6. At a utilization of 3.0 (i.e., when the system is overloaded, as discussed in the workload generation above), DNA's and DADNA's miss rates are only half of vC^{2}M's, and the former remains strictly below the latter as the system becomes increasingly overloaded. Thus, the advantage of DNA/DADNA over vC^{2}M is beyond just the gain in schedulability: with a substantial lower deadline miss ratio, they deliver much better QoS than vC^{2}M, and this matters in a soft real-time context.

D. Latency

Another potential benefit of DNA and DADNA is that, due to the higher throughput, the latency of the jobs is potentially lower. Our next experiment is designed to examine this. We ran the same workloads as before, using vC^{2}M, DNA, and DADNA, but this time we measured the latency of each job. We then computed the average, 99.99^{th} percentile, and worst-case latencies for each algorithm across all workload utilizations.

Figure 5(c) shows our results. The numbers above the columns show DNA's and DADNA's latency reduction factors relative to vC^{2}M. Again, DNA and DADNA substantially outperform vC^{2}M: they cut the average, the 99.99^{th} percentile, and the worst-case latencies by more than half. This is expected: it is well known that EDF produces increasing latencies under

overload conditions, since jobs tend to "back up" for some time once a deadline is missed, causing cascades of additional deadline misses along the way, and this effect increases with utilization. (This is also why the numbers are so high in absolute terms.) Thus, by making the best use out of the resources, DNA and DADNA can effectively reduce not only the average but also the tail and worst-case latencies.

Figure 6 shows these results in more detail; it contains CDFs of the normalized latency (that is, the ratio of latency to deadline) for different workload utilizations: 1.4 (a), 2.0 (b), and 2.6 (c). The results reinforce the earlier point that DNA's and DADNA's more efficient use of the available resources improves latency substantially, relative to vC^{2}M. In this experiment, there is little difference between DNA and DADNA because the behavior of the two differs only very close to a deadline.

	Utilization = 1.0			Utilization = 2.0			Utilization = 3.6		
	Avg	99.99$^{\text{th}}$	Max	Avg	99.99$^{\text{th}}$	Max	Avg	99.99$^{\text{th}}$	Max
DNA	4.5	14.3	13.9	3.2	5.3	5.0	1.7	2.0	2.0
DADNA	4.5	14.8	14.8	3.3	5.6	5.6	2.2	2.4	2.4

Fig. 7: Latency reduction factors relative to vC^{2}M.

The results also show that DNA's and DADNA's improvement factors depend on the system loads. Figure 7 shows their overall latency reduction factors, relative to vC^{2}M, at three different workload utilizations (1.0, 2.0 and 3.6) that represent light load, medium load and heavily overload scenarios. Again, DADNA and DNA consistently outperform vC^{2}M by a significant factor. For example, DADNA and DNA reduce the average latency by more than 4.5× at light load, 3.2× at medium load, and 1.7× at heavily overload scenarios, compared to vC^{2}M. The reduction factors for the 99.99^{th} percentile and worst-case latencies are even more significant: DADNA and DNA cut latencies by more than 14×, 5.3× and 2.0× at light load, medium load and heavily overload scenarios, respectively.

E. DADNA vs. DNA

As shown in Figure 5(c) and Figure 7, DADNA is more effective than DNA in reducing latencies. However, their difference in schedulability and deadline miss ratio in Figures 5(a-b) appears to be somewhat small. This is because both algorithms allocate resources dynamically and efficiently, and because EDF is used in both cases; the only difference is that DADNA can handle an (important) corner case in which a job is not schedulable in the usual way but can be pushed over the edge with a greater resource allocation. This was observed in our results for bimodal tasksets, where DADNA offers up to 1.5× improvement over DNA in schedulability (and latencies); due to space constraints, we omit the details.

F. Summary

By taking phases into account and by reallocating resources dynamically, both DNA and DADNA can use the available resources more effectively than a static allocation method, such as vC^{2}M. This results in better schedulability, lower job deadline miss ratio, and lower latencies. In terms of overall performance, the two algorithms are similar, but DADNA is noticeably better than DNA at reducing latencies.

IX. Conclusion

Our results suggest that it makes sense for schedulers and resource allocators to "look a bit more closely" at the tasks in their workloads. By leveraging an observation from the architecture community – namely that many programs go through multiple phases with distinct characteristics – DNA and DADNA are able to improve the performance of a system *without* adding more resources, by allocating the existing resources more effectively to the tasks that can benefit the most. Compared to prior work, this results in substantially better schedulability and a factor-of-two latency reduction. And yet, the phase analysis we used is relatively simple; to us, it seems likely that there is a lot more information about the needs and behaviors of tasks that could be extracted – e.g., through profiling or static analysis – and used profitably to improve scheduling. This could be an interesting future work. Another interesting direction is to develop a close-formed schedulability test for DNA and DADNA to bring their benefits to hard real-time systems.

Acknowledgments

We thank the anonymous reviewers for their thoughtful comments and suggestions. This work was supported in part by NSF grants CNS-1563873, CNS-1703936, CNS-1750158, and CNS-1955670, and by ONR N00014-20-1-2744.

References

[1] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In *ECRTS*, 2017.

[2] ARM. PrimeCell level 2 cache controller (PL310) - technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0246c/index.html. Accessed: 2015-03-29.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[4] S. Bateni, Z. Wang, Y. Zhu, Y. Hu, and C. Liu. Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform. In *RTAS*, 2020.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[6] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[7] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *RTSS*, 2006.

[8] T. Caliński and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics*, 3(1):1–27, 1974.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contenton on a chip multi-processor architecture. In *HPCA*, 2005.

[10] T. Chen and L. T. X. Phan. SafeMC: A system for the design and evaluation of mode-change protocols. In *RTAS*, 2018.

[11] M. Chisholm, W. Bryan C, N. Kim, and J. H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS*, 2015.

[12] C.-B. Cho and T. Li. Complexity-based program phase analysis and classification. In *PACT*, 2006.

[13] C.-B. Cho and T. Li. Using wavelet domain workload execution characteristics to improve accuracy, scalability, and robustness in program phase analysis. In *ISPASS*, 2007.

[14] C. Courtaud, J. Sopena, G. Muller, and D. G. Pérez. Improving prediction accuracy of memory interferences for multicore platforms. In *RTAS*, 2019.

[15] D. Dasari, V. Nelis, and B. Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Syst.*, 52(3):272–322, May 2016.

[16] D. L. Davies and D. W. Bouldin. A cluster separation measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, 1979.

[17] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *ASPLOS*, 2014.

[18] A. Dhodapkar and J. E. Smith. Dyanmic microarchitecture adaptation via co-designed virtual machines. In *ISSCC*, 2002.

[19] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, 2002.

[20] S. V. Gheorghita, T. Basten, and H. Corporaal. Intra-task scenario-aware voltage scheduling. In *CASES*, 2005.

[21] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[22] D. Guo and R. Pellizzoni. A requests bundling dram controller for mixed-criticality systems. In *RTAS*, 2017.

[23] M. Hähnel and T. Smejkal. Modular energy modeling using energy/utility. In *ICPE*, 2018.

[24] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *RTAS*, 2015.

[25] M. Hassan, H. Patel, and R. Pellizzoni. PMC: A requirement-aware dram controller for multicore mixed criticality systems. *ACM Trans. Embed. Comput. Syst.*, 16(4):100:1–100:28, May 2017.

[26] C.-H. Hsu, U. Kremer, and M. S. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *ISLPED '01*, 2001.

[27] T. Huffmire and T. Sherwood. Wavelet-based phase classification. In *PACT*, 2006.

[28] Intel. Improving real-time performance by utilizing cache allocation technology, Apr. 2015. White Paper.

[29] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B, Nov. 2020. Order No. 325462-073US, https://software.intel.com/content/dam/develop/external/us/en/documents-tps/325462-sdm-vol-1-2abcd-3abcd.pdf.

[30] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.

[31] H. Kim and R. R. Rajkumar. Real-time cache management for multi-core virtualization. In *EMSOFT*, 2016.

[32] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, and F. D. Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *RTAS*, 2017.

[33] N. Kim, B. C. Ward, M. Chisholm, C.-Y. F. , J. H. A. , and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *RTAS*, 2016.

[34] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *ISPASS*, 2005.

[35] B. Li, L. Zhao, R. Iyer, L.-S. Peh, M. Leddige, M. Espig, S. E. Lee, and D. Newell. CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs. *Journal of Parallel and Distributed Computing*, 71(5):700 – 713, 2011.

[36] Y. Li, B. Akesson, and K. Goossens. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Syst.*, 52(5):675–729, Sept. 2016.

[37] J. Lin, Q. Lu, X. Ding, Z. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA*, 2008.

[38] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *ISCA*, 2014.

[39] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *Proc. SOSP*, 2011.

[40] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *RTAS*, 2013.

[41] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *RTNS*, New York, NY, USA, 2015. ACM.

[42] T. K. Moon. The expectation-maximization algorithm. *IEEE Signal Processing Magazine*, 13(6):47–60, 1996.

[43] F. Mueller. Compiler support for software-based cache partitioning. In *LCTES*, 1995.

[44] J. Park, S. Park, and W. Baek. CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *EuroSys*, 2019.

[45] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *RTAS*, Washington, DC, USA, 2011. IEEE Computer Society.

[46] R. S. Peterson and E. G. Sirer. Antfarm: Efficient content distribution with managed swarms. In *NSDI*, 2009.

[47] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[48] D. A. Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, 741, 2009.

[49] M. Roitzsch, S. Wächtler, and H. Härtig. Atlas: Look-ahead scheduling using workload metrics. In *RTAS*, 2013.

[50] Real-time deferrable server (rtds) scheduler. https://wiki.xenproject.org/wiki/RTDS-Based-Scheduler.

[51] S. Schliecker and R. Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, Jan. 2011.

[52] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, 2010.

[53] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, 2011.

[54] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. Mete: Meeting end-to-end QoS in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev.*, 39(1):13–24, June 2011.

[55] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, 2004.

[56] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report CS99-630, UCSD, Aug. 1999.

[57] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, 2001.

[58] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, 2002.

[59] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA*, 2003.

[60] Splash2x benchmark. http://parsec.cs.princeton.edu/parsec3-doc.htm#splash2x.

[61] L. Subramanian. *Providing High and Controllable Performance in Multicore Systems Through Shared Resource Management*. PhD thesis, Carnegie Mellon University, 2015.

[62] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[63] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *ICESS*, 2013.

[64] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS*, 2016.

[65] X. Wang and J. F. Martinez. Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *HPCA*, 2015.

[66] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36, May 1995.

[67] J. Xiao, S. Altmeyer, and A. D. Pimentel. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *RTSS*, 2017.

[68] Y. Xie and G. H. Loh. Dynamic classification of program memory behaviors in CMPs. In *CMP-MSI*, 2008.

[69] M. Xu, R. Gifford, and L. T. X. Phan. Holistic multi-resource allocation for multicore real-time virtualization. In *DAC*, page 168, 2019.
[70] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic resource allocation for multicore real-time systems. In *RTAS*, 2019.
[71] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. vcat: Dynamic cache management using cat virtualization. In *RTAS*, 2017.
[72] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751, Sept 2016.
[73] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *PACT*, 2014.
[74] Y. Ye, R. West, J. Zhang, and Z. Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *RTSS*, 2016.
[75] H. Yun, R. Pellizzoni, and P. K. Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *ECRTS*, 2015.
[76] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS*, 2012.
[77] 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 *RTAS*, 2013.
[78] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016.
[79] Y. Zhou and D. Wentzlaff. MITTS: Memory inter-arrival time traffic shaping. In *ISCA*, 2016.
[80] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contentionin multicore processors via scheduling. In *ASPLOS*, 2010.