Providing Task Isolation via TLB Coloring

Shrinivas Anand Panchamukhi
North Carolina State University
Department of Computer Science
Raleigh, NC, USA
spancha@ncsu.edu

Frank Mueller
North Carolina State University
Department of Computer Science
Raleigh, NC, USA
mueller@cs.ncsu.edu

***Abstract*—The translation look aside buffer (TLB) improves the performance of systems by caching the *virtual page to physical frame* mapping. But TLBs present a source of unpredictability for real-time systems. Standard heap allocated regions do not provide guarantees on the TLB set that will hold a particular page translation. This unpredictability can lead to TLB misses with a penalty of up to thousands of cycles and consequently intra- and inter-task interference resulting in loose bounds on the worst case execution time (WCET) and TLB-related preemption delay.**

In this work, we design and implement a new heap allocator that guarantees the TLB set, which will hold a particular page translation on a uniprocessor of a contemporary architecture. The allocator is based on the concept of *page coloring*, a software TLB partitioning method. Virtual pages are colored such that two pages of different color cannot map to the same TLB set.

Our experimental evaluations confirm the unpredictability associated with the standard heap allocation. Using a set of synthetic and standard benchmarks, we show that our allocator provides task isolation for real-time tasks. To the best of our knowledge, such TLB isolation without special hardware support is unprecedented, increases TLB predictability and can facilitate WCET analysis.

I. Introduction

The translation look aside buffer (TLB) is a hardware cache that sits close to the processor and caches the *virtual page* to *physical frame* mappings. Today's computer architectures support multiple levels of TLBs. Most Intel [11], AMD [2] and ARM [3] processors feature separate instruction (ITLB) and data (DTLB) TLBs at level one (L1). In some processors [11], the level two (L2) of the TLB is shared between data and instructions while in others [2], the DTLB is shared between different cores. Typically, a virtual address is translated to the physical address using a hierarchy of translation tables known as *paging structures*.

Static timing analysis tools [25] analyze the source code of a task and compute the worst case execution time (WCET) of the task. Mueller [17] (among others) has proposed abstract cache states to analyze instruction caches to achieve tight bounds on WCET. White et al. [24], Ramaprasad et al. [20] [21] and Ferdinand et al. [8] have bounded the WCET for systems with data caches. TLBs have only been considered in the context of achieving tight bounds on the WCET of a task with special compiler [18] or hardware TLB locking [12] support, but not purely in software without compiler modifications.

TLBs improve the performance of the system in terms of speeding up the page translations, but they are also a source of unpredictability for real-time systems. In this work, we focus on the TLB effect of data references to the heap. Historically, heap allocation was avoided in real-time systems due to the following unpredictability issues. First, the WCET of the standard heap allocation API is too high or is unbounded [16]. Second, standard heap allocation APIs do not provide the means to specify a DTLB set in which the virtual page to physical frame mapping will be placed. Our focus in this paper is on the latter point. This unpredictability poses three problems.

First, it makes it difficult for static timing analysis to determine whether a memory access will hit or miss in the DTLB. Hence, for calculating the WCET of a task, static timing analysis would need to explicitly model TLB cache behavior per reference for intra-task behavior. In addition, TLB-related preemption delays across tasks would need to be modeled, either by assuming full TLB invalidation at context switches or by explicitly modeling the subset of pages in inter-task TLB analysis and depending on whether the operating system (OS) performs a partial or full TLB flush. The pessimism of such TLB modeling leads to inefficient processor utilization.

Second, heap allocation may utilize the DTLB sets in a non-conflict aware manner. For example, consider two tasks, T1 and T2, and a system with a 2-way set associative DTLB. Further assume that the DTLB is empty. Then let each task request two virtual pages from the standard heap allocator. The heap allocator may allocate virtual pages in such a way that all four pages map to the same DTLB set. Note that the DTLB was initially empty and had sufficient space to accommodate all four mappings without conflicts.

Third, if T1 and T2 conflict in the DTLB, they might evict each other's mappings in the DTLB repeatedly. If tasks T1 and T2 are hard real-time (HRT) and soft real-time (SRT) tasks, respectively, the HRT task will suffer interference from the SRT task. This interference could result in the HRT task missing its deadline unless the DTLB were explicitly modeled. But even if the DTLB were modeled, it might result in loose WCET bounds due to (1) conservative overestimations of intra-task TLB misses combined with (2) overestimations of inter-task TLB-related preemption delay, where the number of preemption points is also hard to bound [19].

In this paper, we describe the design and implementation of a new heap allocator that we refer to as *tlb_malloc*. This allocator utilizes the concept of page coloring to provide TLB partitioning in software. We assign colors to virtual pages in such a way that virtual pages with different color will not map to the same DTLB set. When tasks dynamically allocate memory using *tlb_malloc*, in addition to specifying the size of the allocation, they will also specify the color of the memory region. By ensuring that each task allocates memory regions of a unique color, we can provide isolation between tasks and thereby obviate the need for inter-task TLB analysis. If sufficient colors exist per task, we no longer need to consider intra-task TLB conflicts in WCET bounds either, which may lead to significantly tighter bounds on the WCET. Our experimental evaluations reveal the unpredictability associated with standard heap allocation. Using a set of synthetic and standard benchmarks, we show that tasks can be isolated from each other, i.e., no inter-task DTLB misses are incurred.

Past research has utilized page coloring to guarantee task isolation with respect to caches and DRAM. Ward et al. [23] have proposed coloring physical frames such that two frames with different colors will not cause last-level cache conflicts. Yun et al. [26] have proposed a DRAM bank-aware memory allocation. This ensures that concurrently running applications on different cores do not access memory that maps to the same DRAM bank. Our work focuses on uniprocessor TLBs and directly applies to private DTLBs of contemporary multicores. Section VII discusses how it can be extended to shared DTLBs in multicores.

Our contributions in this paper are:

1) We design and implement a new heap allocator that provides guarantees on the DTLB sets that hold a particular virtual page to physical frame mapping.
2) We devise experiments to assess which bits of the virtual address determine the DTLB set, what the cost of a single TLB miss is and how much OS interference exits.
3) We conduct experimental evaluations of the heap allocator to demonstrate task isolation.

The rest of the paper is organized as follows: Section II discusses related work. A generic design of our allocator is presented in Section III. Section IV describes the implementation details of *tlb_malloc*. Section V discusses the experimental framework. Section VI presents the results for both synthetic and standard benchmarks. Section VII summarizes the contributions and discusses open problems.

II. Related Work

Providing a virtual address space for safety critical systems with different integrity levels has been explored by Bennett and Audsley [4]. The main goal of their work was to provision virtual memory for safety critical systems without complicating timing analysis for tasks with hard deadlines. To support the concept of a *real-time address space*, kernel modifications are imperative. Their paper, however, neither considered tasks that use dynamic memory allocation nor the interference due to TLB sets shared by tasks. Our approach is different in the sense that no kernel modifications are needed and real-time tasks can use dynamic memory allocation with bounded or even without TLB interference.

Puaut et al. [18] have proposed a compiler approach to make paging more predictable. The main idea is to identify *page in* and *page out* points of virtual pages at compile time. This method relies on the static knowledge of the possible references between virtual pages of tasks. However, the focus of their paper is to make *demand paging* more predictable while ours is on task isolation with respect to TLB.

Compiler-directed page coloring proposed by Bugnion et al. [6] involves three key phases. First, a compiler creates a summary of array references and communicates this information to the runtime system. The runtime system then uses machine-specific parameters (e.g., cache size) to generate a preferred color for the physical frame. The OS then uses this color as a hint in a best effort attempting to honor them. This technique is applicable to physically indexed caches while our focus is on TLBs caching virtual page translations. In addition, our technique does not require profiling or modifications to the compiler and the OS.

Software cache partitioning is related to our idea. This is commonly known as cache coloring. The main idea of this technique is to color physical frames such that two frames of different color will not map to the same cache set. Liedtke at al. [13] propose OS-controlled cache partitioning. Mancuso et al. [15] use memory profiling to identity hot pages in virtual memory. Then, the kernel subsequently allocates physical frames to these pages such that there are no cache conflicts. Ward et al. [23] proposed cache locking and cache scheduling for the last-level caches. Their scheme treats cache ways as resources that must be acquired by tasks before they are activated. These techniques alleviate interference with respect to physically indexed caches but do not consider the interference in the TLB.

TLSF [16] is an approach to support dynamic memory allocation and deallocation in constant time for real-time systems. CAMA [10] builds upon TLSF to incorporate cache awareness. CAMA can allocate dynamic memory in constant time and can also guarantee the cache set that will hold this allocated memory. PALLOC [26] is a DRAM bank-aware memory allocator. It ensures that tasks running concurrently on different cores do not access physical memory that maps to the same DRAM bank. Thus, PALLOC reduces DRAM bank-level interference between tasks. Though these techniques enable real-time tasks to use dynamic memory, they do not consider the interference they may cause in the TLB. In contrast, our allocator focuses on providing isolation between tasks with respect to the TLB. Our TLB allocator operating on virtual pages can be complemented by the techniques of TLSF, CAMA and PALLOC, which operate on physical frames.

Ishikawa et al. [12] study the benefits of architectural support to lock TLB entries provided by some ARM platforms. A static locking scheme for TLB entries of real-time tasks provides little overhead but makes non-real-time tasks significantly less efficient. Dynamic TLB locking with invalidation of locked entries around real-time tasks and another scheme with invalidation of all entries result in moderate runtime overhead for fewer and more locked pages, respectively. This method relies on hardware TLB locking support. In contrast, our method provides similarly predictable TLB miss behavior in the absence of hardware support for TLB locking. Furthermore, their work assumes non-preemptive real-time tasks while we study the more general problem of preemption among real-time tasks. Their and our approach share the idea of TLB partitioning, albeit the former via hardware capabilities while the letter does so in software, which makes it more widely applicable.

III. Design

To describe our heap allocator, we start with a basic design and then incrementally add complexities and design details. Consider an N-way set associative DTLB supporting $N \times M$ entries and a virtual address space with $N \times M$ pages as shown in Figure 1. The DTLB handles translations for virtual pages at page size (*pg_size*) granularity. Our objective is to provide task isolation by TLB partitioning via coloring. In other words, pages map into TLB entries, which are colored. By controlling a page address upon heap allocation, one can ensure that a set of pages of a task has color(s) disjoint from those of any other task. Thus, there cannot be any inter-task TLB conflict misses since pages of different tasks belong to disjoint TLB sets. Furthermore, if the number of pages of a given color (set) does not exceed the TLB associativity (number of ways), then no intra-task TLB conflict misses can occur, only cold misses upon a first reference will be incurred during initialization.

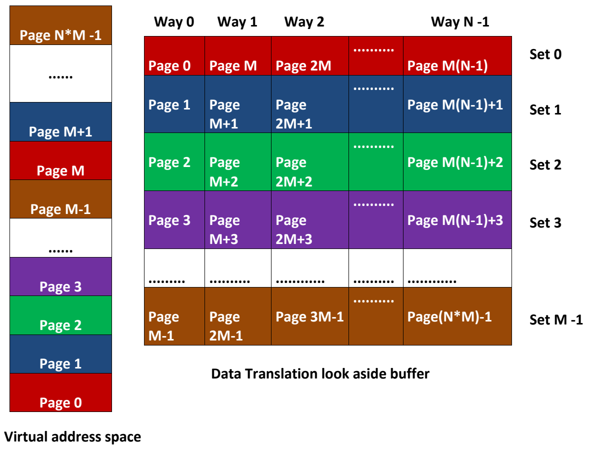

Fig. 1: TLB coloring - max contiguous allocation = *pg_size*

Let us assume that page 0 will map to set 0 as shown in Figure 1. The translation for page 0 could be stored in any one of the N ways. For simplicity, let us assume that the DTLB is initially empty. We fill the entries from left to right in each set. Further, assume that page 1 will map to set 1 and so on until page $M-1$. Pages M to $2M-1$ will wrap around, i.e., page M will map to set 0, page $M+1$ will map to set 1 etc.

We color pages 0, M, $2M$ ... $M(N-1)$ with the same color (red in this example) because all of them map to the same DTLB set. Similarly, pages 1, $M+1$, $2M+1$ ... $M(N-1)+1$ are colored blue and so on. We can see that no two virtual pages with different color can map to the same DTLB set. Each DTLB entry holds a translation for a virtual page of size *pg_size*. Since each DTLB set is given one color, the maximum contiguous virtual address space one can allocate of a particular color is given by the *pg_size*.

Let us now consider that a task needs to allocate more than *pg_size* bytes of contiguous memory in the virtual address space. This typically results in allocating arrays that span across multiple pages. In our basic design, such an allocation would span across two pages of different color assuming that the allocation is aligned to a page boundary. In order for the tasks to be able to allocate contiguous virtual memory of size greater than *pg_size* of a particular color, we reserve a certain number of DTLB sets for contiguous memory allocation and require the number of multi-page allocation over *all* tasks to be constrained by the number of ways N. Consider the same example as in Figure 1 but with R sets reserved for contiguous memory allocation. Figure 2 shows an example of DTLB coloring that supports greater than *pg_size* allocations for a particular color.

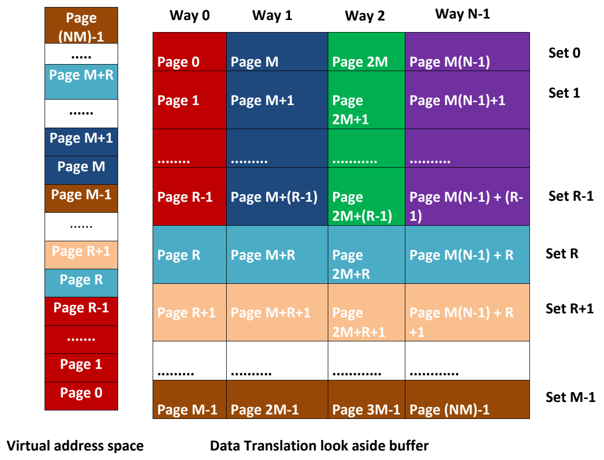

Fig. 2: TLB coloring - max contiguous allocation = $pg\_size \times R$

Since we have R sets reserved for contiguous allocations, the maximum amount of contiguous memory that can be allocated of a particular color is $R \times pg\_size$. Pages 0 to $R-1$ are colored red. Starting from page R until page $M-1$, the coloring is performed in a similar way as described for the basic design. But pages M to $M+(R-1)$ are colored blue, again to support contiguous memory allocation. All the other virtual pages are colored in a similar manner. Notice that multi-page allocations are still isolated from each other. Since we require the number of multi-page allocations to not exceed N, no TLB misses will occur (after a single, initial cold miss) per page.

Contemporary operating systems support the concept of huge pages whose size varies from 2 MB to 1 GB depending on the configuration and the system in place. Computer architectures have a different DTLB structure to support huge page translations. Our heap allocator handles huge page allocation in a similar manner as described above.

We now generalize the methods for DTLB coloring described so far. Let M be the total number of DTLB sets, R be the number of sets reserved for contiguous allocations and N be the associativity of the DTLB. Let *pg_size* denote either the normal page size or huge page size of the system.

Corollary 1. *The number of colors for allocations greater than pg_size is equal to N.*

Proof: Follows from the definition of N. ■

Let *max_alloc_size* refer to the maximum contiguous allocation size that can be served by our allocator. Let *num_colors_up_to_pg_size* refer to the number of colors available for allocations not exceeding *pg_size*. Then,

$$\text{max\_alloc\_size} = \text{page\_size} \times R, \tag{1}$$

$$\text{num\_colors\_up\_to\_pg\_size} = M - R. \tag{2}$$

The number of instances of a particular color is constrained by the associativity of the DTLB.

IV. Implementation

In this section, we describe the notation and the data structures to implement our heap allocator. We then present generic algorithms, which can be implemented on any architecture. Finally we present a specific instance of the generic algorithm that we have implemented on an Intel Xeon E-2650.

A. Notations

We refer to the routine that initializes our heap allocator as *tlb_malloc_init*, the heap allocator as *tlb_malloc* and the deallocator as *tlb_free*. These three routines are exposed as library functions to user space applications. Compared to the standard heap allocation API, *tlb_malloc* and *tlb_free* take an additional parameter, *color*, indicating the color of the memory region to be allocated. For each DTLB to be colored, *tlb_malloc_init* sets aside a virtual address space of $pg\_size \times dtlb\_sets$ bytes. Additional memory may be needed to handle page boundary alignment. Our heap allocator serves allocations from this virtual address space pool that is set aside.

Let us provide a generic description of the internals of *tlb_malloc* and *tlb_free*. It is applicable to all the DTLBs to be colored. Depending on the number of bytes requested, *tlb_malloc* will call one of the functions as shown in Figure 3. *LEN_BYTES* refers to the number of bytes the allocator uses to store the size of the allocation. Similarly, *tlb_free* will call one the functions shown in Figure 4 depending on the size of the allocation referenced by *ptr*.

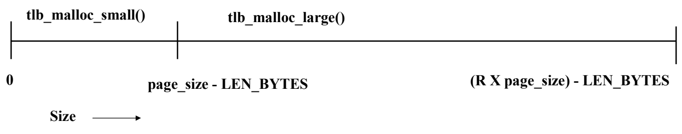

Fig. 3: tlb_malloc internals

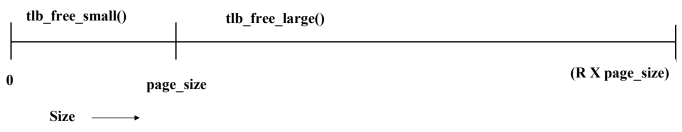

Fig. 4: tlb_free internals

B. Data structures

We utilize a singly-linked *free block* data structure with a length attribute that is generic as it is not tied to any specific language or implementation. In addition, we maintain a *free list* per color to track the available memory. Figure 5 shows a generic representation of the *free list*. *Color 1 pointer, Color 2 pointer, ... Color n pointer* are the base pointers, which point to the first available *free block* of that particular color. Each *free block* in turn points to the next available *free block* of the same *color*. We need two such *free lists*, one for *small* allocations and the other for *large* allocations, for each DTLB to be colored.

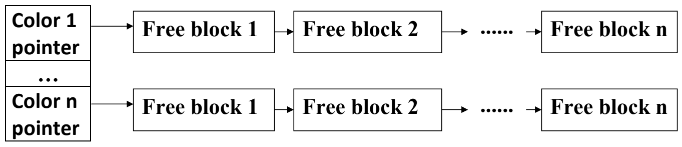

Fig. 5: Data structures

C. Generic algorithms

The different heap allocation APIs from Figure 3 are internally mapped onto a single core allocator with prototype `allocate_memory(type,size,color)`, i.e., *type* identifies the the allocation routine (see Figure 3) and free lists are subsequently specialized per type and *color*. The allocator traverses this specific *free_list* to find a suitable memory block and returns its address. It also stores the allocation size (hidden in memory preceding the returned address). The heap deallocator with prototype `free_memory(type,ptr,color)` adds the memory block (of the internally stored length) back to the appropriate *free_list* of a certain type (for all free APIs from Figure 4).

Algorithm 1 shows how to ascertain the bits in the virtual address that determine the DTLB set, which may not always be documented but also helps to pin-point OS interference to specific sets discussed later. The algorithm relies on hardware performance counters monitored via the library PAPI [7]. Let *no_entries_in_dtlb* be the number of entries supported by the DTLB, *dtlb_assoc* be the associativity of the DTLB and *pg_size* denote the page size on the system. The algorithm takes an array *page_numbers* as a parameter. Each element of the array indicates the page number to be accessed. Pages are numbered starting from zero. Line 5 allocates a memory region large enough to contain $no\_entries\_in\_dtlb + 1$ pages. In this algorithm, *data* is of the type integer but any other data type could also be used. The *conflict_page* value (line 6) is used to index into *data*. This value represents the index of the first element in the conflicting page. To explain the main idea of the algorithm, we consider the following example. On the Intel Xeon E-2650, the L1 DTLB is 4-way set associative supporting 64 entries. This DTLB handles 4kB translations. The *conflict_page* value is 64×1024 in this case. We keep this value constant and vary *page_numbers* for all possible permutations of 64 pages. Lines 9-12 access the pages as indicated by the *page_numbers* array. In this algorithm, we perform a write operation (but a read operation could also be used to trigger DTLB accesses). By reading the hardware performance counters via PAPI (lines 7+16), we determine the combination of 5 pages that result in a DTLB miss on every memory access in the outer loop (assuming a LRU / PLRU replacement policy). Thus, knowing the set of pages that conflict in the DTLB, we can examine the virtual address of these five pages to determine the common bits after stripping the page offset bits from the virtual address. After every memory access in the outer and inner loop, we need a memory fence instruction to ensure that the memory references are issued in order and are recorded by the performance monitoring registers. This algorithm is later also used to determine the cost of a single TLB miss and analyze interference with OS activity.

Algorithm 1 Find address bits which determine DTLB set

```
function FIND_BITS(page_numbers[dtlb_assoc])
    n = 1000;
    no_entries_in_dtlb = $no\_of\_dtlb\_sets \times dtlb\_assoc$;
    $int\_elements = pg\_size/sizeof(int)$;
    int *data=malloc(($no\_entries\_in\_dtlb$+1)$\times$ $pg\_size$);
    conflict_page=$no\_entries\_in\_dtlb \times int\_elements$;
    PAPI_read();
    for $i = 0$ to $n$ do
        for $j = 0$ to dtlb_assoc do
            data[ $page\_numbers[j] \times int\_elements$] = 2;
            asm("memory fence instruction;");
        end for
        data[ $conflict\_page \times int\_elements$] = 2;
        asm("memory fence instruction;");
    end for
    PAPI_read();
end function
```

V. Experimental framework

In this section, we describe the experimental framework used to evaluate our new heap allocator. We implement periodic releases of a task set using POSIX Threads and timers to enforce phases and periods of real-time tasks. We conducted experiments on an Intel Xeon E5-2650 processor under Linux 2.6.32. The system has 16 cores running at up to 2 GHz. The L1 DTLB is 4-way set associative with 64 entries. The L2 TLB supports 512 entries and is 4-way set associative. The L2 TLB is shared between instruction and data. Both these TLBs handle 4kB sized page translations and are private to a single core. For all experiments, we use rate monotonic scheduling of a set of tasks on a single core. On Linux, this can be achieved by setting the task priority while creating them. We use the *SCHED_FIFO* real-time scheduling policy and bind all tasks to a particular core to avoid task migration.

Given the 4-way set associative L1 DTLB with 64 entries of the Xeon E-2650, from Corollary 1 we know that the value of N is 4. The parameter *R* is configurable and is passed as an input to *tlb_malloc_init*. The values for Equations 1 and 2 and are determined at runtime. In our implementation, the allocation structure is the same as described above. For the *free list* per color, we maintain an array of pointers. Each index in the array refers to a particular color and the value at that index points to the first available block of free memory of that particular color.

VI. Results

In this section, we first assess the predictability of *malloc* and *tlb_malloc*. We then present our approach to identify which of the DTLB sets are subject to OS noise, as defined in Section VI-B. Further, we estimate the L1 and L2 DTLB miss penalties. We then evaluate the performance of our allocator with a set of synthetic benchmarks and standard benchmarks.

A. Predictability

In this experiment, we create two tasks, T1 and T2. Each task allocates 32 pages. From the pool of 64 pages, the number of page mappings to each DTLB set is recorded. We then compute the maximum and average mappings per DTLB set over five runs. Figure 6 shows the graph of the DTLB set versus the number of mappings. *malloc_max* represents the maximum number of mappings that was observed when both tasks use *malloc* to allocate their pages. Similarly, *malloc_avg* represents the average number of mappings. *tlb_malloc_max_avg* represents both the maximum and average number of mappings when both tasks used *tlb_malloc* to allocate their pages. Since the values are identical, we just use one line to represent both the cases.

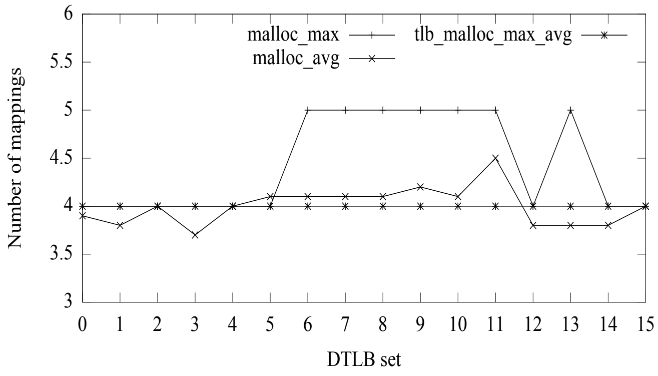

Fig. 6: DTLB set utilization

Considering the L1 DTLB with its 4-way set associative configuration, the best possible DTLB set utilization would be one in which each DTLB set has four mappings. In Figure 6, *malloc_max* shows that DTLB sets 0-5, 12, 14 and 15 have four mappings while DTLB sets 6-11 and 13 have five mappings. *malloc_avg* shows that the average number of mappings per DTLB set varies from 3.7 to 4.5. On the other hand, *tlb_malloc_max_avg* shows each DTLB set to have exactly four mappings. From Figure 6, we can conclude that *tlb_malloc* results in higher predictability compared to *malloc*. Our allocator is predictable in the sense that it guarantees the DTLB set that will hold a particular page translation. Thus, if tasks T1 and T2 use *malloc* to allocate pages, they might interfere with each other in the DTLB. But if T1 and T2 use *tlb_malloc*, such inter-task interference can be avoided.

B. Identifying noisy DTLB sets

So far, we have assumed that the entire DTLB is available for a set of tasks. But when the tasks run, they might invoke certain library functions. These library calls and OS activities, such as scheduling, interrupts, multi-level page tables etc., make use of the DTLB. We call this "OS noise". Hence, only a certain number of ways of a DTLB set can be used by the tasks without noise. The following describes our methodology to identify the number of ways that can be used by tasks beyond those already occupied due to OS noise.

tlb_malloc gives us control over the DTLB set that will hold a particular page translation. Hence, we use *tlb_malloc* to identify the number of ways of a DTLB set that are available for our experiments. We turn off address space randomization (a security feature of Linux) so that the stacks of the tasks always have the same base address across multiple runs. We start with two tasks, where each task allocates one page of a particular *color*. Each task has a *warm-up phase* and a *repeated access phase*. When the tasks run, we should not observe any misses apart from the *warm-up phase*. However, if one of the two tasks or both tasks incur misses, then we change the *color* of the allocation until both tasks do not incur any misses anymore. Further, the number of pages accessed by each task can be increased. Using this methodology, we can systematically distinguish those DTLB sets subject to OS noise from clean ones available for a task set. We use this methodology for experiments with *tlb_malloc* to ensure that we do not use any noisy DTLB sets. For experiments using *malloc*, we cannot perform such identification because *malloc* does not give us control over the DTLB set that will hold a particular mapping.

C. L1/L2 DTLB miss penalty

In this experiment, we determine the DTLB miss penalty for the Intel Xeon E-2650. We measure the execution cycles using the *time stamp counter* register. To calculate the L1 DTLB miss penalty, we map five pages to the same DTLB set using our allocator. We perform a warm-up by accessing the pages once. We then access these five pages in a cyclic manner similar to the one shown in Lines 8-14 of Algorithm 1. This results in a L1 DTLB miss on every access. The average latency is computed by dividing the execution cycles by the number of L1 DTLB misses. This latency also factors in the cycles required for *memory fence instructions* and the overhead of the memory reference itself. To factor out this overhead via dual loop timing [1], we run another loop in which we access elements of an array within one page in the similar manner as we access the five pages. Subtracting this overhead from the average latency calculated closely approximates the L1 DTLB miss penalty. Averaging over 10 runs, we determined the L1 DTLB miss penalty to be about 7.5 cycles. The standard deviation of the results was about 0.04.

To determine the L2 DTLB miss penalty, we use the same idea but map pages to different DTLB sets. We measure the latency of the warm-up loop and compute the L2 DTLB miss penalty in a similar manner as described above. By averaging over 10 runs, we found the L2 DTLB miss penalty to be about 1731 cycles with a standard deviation of 231.

D. Synthetic benchmarks

In the synthetic benchmarks, each task allocates and accesses a certain number of pages. During initialization, each task accesses its set of pages for the first time. We refer to this as the *warm-up phase*. When a job of either task is released, the job accesses its set of pages repeatedly. We refer to this as the *repeated access phase*. With the 4-way set associative L1 DTLB of the Intel Xeon, the worst-case scenario occurs when the combined accesses of both tasks are greater than four pages and these pages map to the same DTLB set. The best-case scenario occurs when the pages of each task map to different DTLB sets. We discuss two variants for this experiment.

1) Same set vs. different set using tlb_malloc: We use the following task set, where each task is denoted as (phase, period, execution): T1 (1ms, 2ms, 0.392ms), T2 (0ms, 16ms, 7.88ms). This task set ensures that the lower priority task suffers more than one preemption. In this experiment, we compare the number of DTLB misses that each task incurs in the best- and the worst-case scenario. For the worst case, both tasks request four pages each from *tlb_malloc* such that all pages are of the same color. This scenario is termed *same set*. Conversely, for the best case, T1 allocates four pages of *color 1* and T2 allocates four pages of *color 2*. This scenario is termed *different set*.

Figure 7 depicts the results for this experiment over two hyperperiods for the the number of DTLB misses (y-axis) over time in milliseconds (x-axis). The graphs for T1 and T2 are shown in the top half and bottom half of the figure, respectively. Each data point represents a job. The time instant before 0ms represents the *warm-up phase*. The corresponding value on the y-axis indicates the number of DTLB misses for the *warm-up phase*. We refer to this as the 0th job of a task.

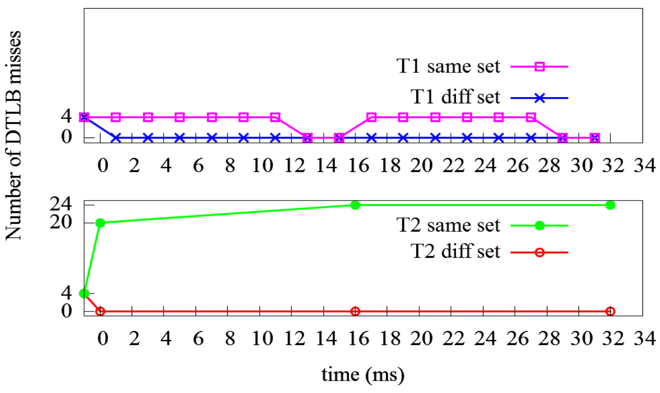

Fig. 7: Same set vs. diff set using tlb_malloc

The misses for the worst-case scenario are depicted as *T1 same set* and *T2 same set*. Jobs 1-5 (released at times 1ms, 3ms, 5ms, 7ms, 9ms) of T1 incur four misses because these jobs preempt T2 and will need to reload its pages into DTLB. Job 6 of T1 incurs four misses because this job is released after T2 completes its execution and will also need to reload its pages into the DTLB. Jobs 7 and 8 of T1 have zero misses because when these jobs are released, T2 has finished its execution so that these jobs do not have any interference. This behavior repeats for each hyperperiod.

Every job of T2 is preempted five times. Hence, each job will incur 20 misses. The job in hyperperiod one has 20 misses while the jobs in other hyperperiods have 24 misses. This is because the job in hyperperiod one begins execution right away after the warm-up. Jobs in other hyperperiods will need to load the pages into the DTLB first because the last job of T1 in the previous hyperperiod would have replaced T2's mappings in DTLB.

The misses for the best case are represented by *T1 diff set* and *T2 diff set*. In the first hyperperiod, even though jobs 1-5 of T1 preempt T2, neither T1 nor T2 incur any misses because both map to different DTLB sets. The misses observed are due to the *warm-up phase*. From Figure 7, we can conclude that if tasks use different DTLB sets then they can be isolated from each other and will not be subject to inter-task interference.

Figure 8 depicts the number of cycles needed by each job of T1 over five hyperperiods. *same set* represents the experiment when both tasks share a DTLB set. *diff set* represents the experiment when the two tasks do not share DTLB sets. For both runs, the jobs in hyperperiod one seem to require a larger number of cycles to execute compared to the jobs in other hyperperiods. We attribute this to the warm-ups in the architecture (e.g., instruction caches, data caches, branch predictors etc.). The execution cycles for jobs in the second hyperperiod for the *same set* experiment varies depending upon the interference from T2. Jobs 9-13 preempt T2. Since these jobs of T1 have to reload their pages, they will incur an extra cost in terms of L1 DTLB miss penalty. Job 14 executes just after T2 completes and will also need to reload its pages and, hence, incurs additional execution cycles. Jobs 15 and 16 are not subject to any interference from T2 and, hence, require lower cycles to complete their execution. *diff set* shows the execution cycles for jobs of T1 in the second hyperperiod to be consistent at about 782566 cycles. On average, the difference between the execution cycles of jobs 9-13 in *same set* and *diff set* is about 30 cycles, which is roughly the L1 DTLB miss penalty for 4 pages. This behavior is repeated for subsequent hyperperiods. From Figure 8, we further confirm that the tasks can be isolated from each other if they use different DTLB sets.

The execution cycles for jobs of T2 also include the cost of preemption along with cycles needed by the OS scheduler. Since we do not use a real-time OS, the scheduling activities are not bounded and may take a variable amount of time. Since the measurements of execution cycles for jobs of T2 are subject to noise, it is difficult to attribute the variation in the execution cycles directly to the DTLB interference. Hence, we do not discuss the execution cycles for T2.

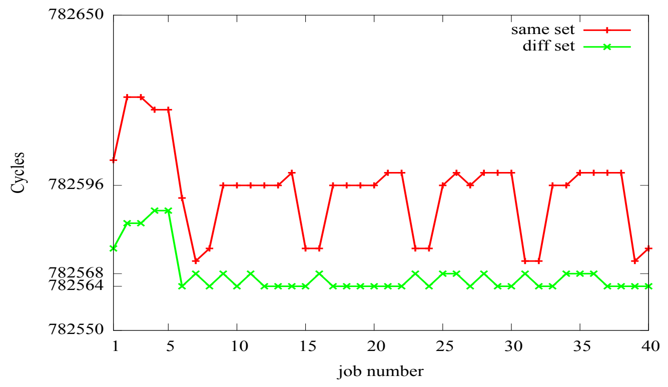

Fig. 8: Jobs vs. execution cycles for T1

2) malloc vs. tlb_malloc: We use the following task set: T1 (1ms, 2ms, 0.385ms), T2 (0ms, 16ms, 7.38ms). In this experiment, each task initially allocates 32 pages so that the execution times differ slightly from the previous task set. Each task then randomly chooses 14 pages from the pool of 32 pages. The tasks then perform a write operation on these 14 pages. Each task consists of a *warm-up phase* and a *repeated access phase* as described previously. For both tasks, we compare the number of DTLB misses when the tasks use *malloc* versus *tlb_malloc* to allocate its pages. For Figures 9-12, the axes and data points have the same meaning as described for Figure 7.

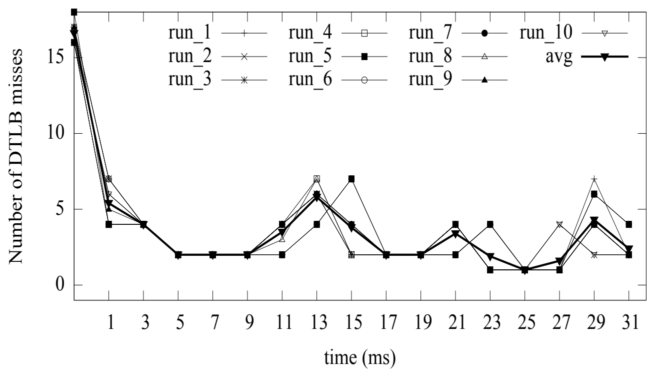

Fig. 9: Malloc: Jobs vs. DTLB misses for T1

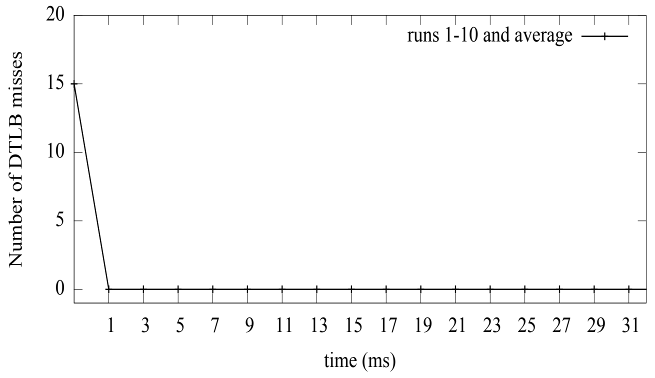

Fig. 10: tlb_malloc: Jobs vs. DTLB misses for T1

Figure 9 depicts the misses (averaged over 10 runs) incurred by T1 when it uses *malloc*. The number of DTLB misses varies with each run depending on the interference with T1. In hyperperiod one, jobs 1 and 2 of T1 have 5 misses on average, jobs 3-5 of T1 have about 2 misses, jobs 6 and 8 have about 3.5 misses and job 7 has about 6 misses. Similar values are observed for hyperperiod two. Figure 10 shows the misses incurred by T1 when it uses *tlb_malloc*. The number of misses for all 10 runs are identical and, hence, we use a single line to represent all 10 runs and the average. In Figure 10, the observed misses of job 0 are due to the *warm-up phase*. Subsequent jobs of T1 do not incur any DTLB misses at all.

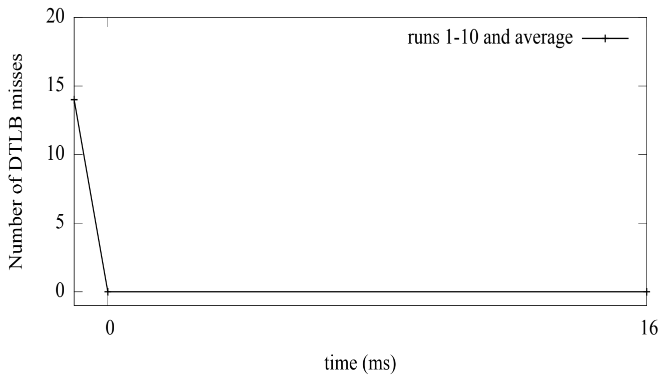

Fig. 12: tlb_malloc: Jobs vs. DTLB misses for T2

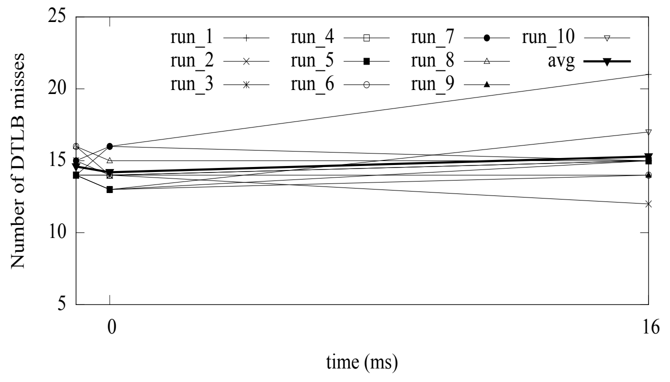

Fig. 11: Malloc: Jobs vs. DTLB misses for T2

Figures 11 and 12 show the results for T2 for *malloc* and *tlb_malloc*, respectively. The results are similar to T1. In Figure 11, the number of misses for *malloc* varies with each run with an average number of misses of 15. In Figure 12, we again use a single line to represent the misses for all 10 runs and the average for *tlb_malloc* since the values are identical. The number of misses observed for job 0 are due to the *warm-up phase* while subsequent jobs do not incur any misses. From Figures 9-12, we conclude that *tlb_malloc* does provide task isolation with respect to the DTLB. These figures also show that the DTLB misses are predictable when the tasks use *tlb_malloc*.

We first conducted experiments to capture the runtime in cycles of T1 in this experiment. These initial experiments (figures omitted) resulted in execution bands reflecting different TLB behavior but variations within each band. We hypothesized that interference between references in the L1/L2 caches were causing this behavior. This was confirmed as follows. We slightly modify the experiment described above. Instead of accessing the first four bytes of each page, we access bytes such that the memory accessed maps to different L1 data cache sets. By this modification, we ensure that we do not incur conflicts in the L1 data cache. The results are depicted in Figure 13.

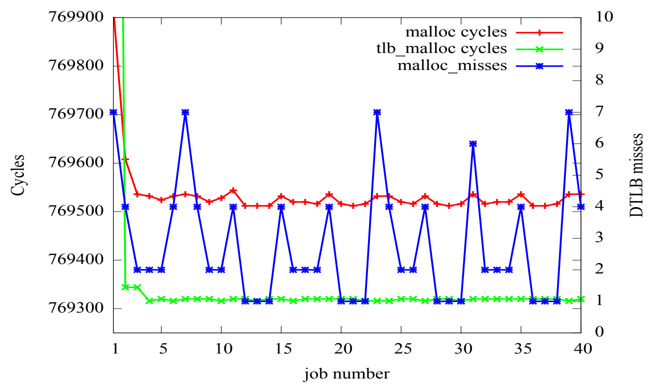

Fig. 13: Jobs vs. execution cycles for T1

In Figure 13, *malloc cycles* and *tlb_malloc cycles* represent the number of cycles (primary y-axis) and TLB misses (secondary y-axis) of each job of T1 (x-axis) when the tasks use *malloc* and *tlb_malloc*, respectively. *malloc_misses* represents the number of DTLB misses incurred by each job of T1 when the tasks use *malloc*. The number of DTLB misses is zero when the tasks use *tlb_malloc* (omitted from the graph). When the tasks use *malloc*, each job executes for a varying number of cycles depending on the DTLB interference caused. But if the tasks use *tlb_malloc*, we observe that the execution time for each job is almost constant. For both *malloc* and *tlb_malloc*, the first job requires a longer time to execute than the other jobs, which is again attributed to a warm-up in the architecture (e.g., instruction caches, branch prediction etc.). Figure 13 further confirms that our allocator does provide task isolation. Due to the same reasons discussed in Section VI-D1, we do not measure the execution cycles for T2.

E. *Malardalen benchmarks*

For the experiments described in this section, we use benchmarks from the Malardalen suite [14] to show task isolation. The experimental setup is similar to that described in Section VI-D2. We modified the benchmarks so that they use heap allocated regions instead of statically allocated ones.

Table I shows the characteristics of tasks for various experiments. Phase, period and execution time are in milliseconds. Column 1 shows the number of tasks in an experiment. The tasks are depicted in decreasing order of priority. In an experiment of j tasks, each task allocates $\left\lceil \frac{64}{j} \right\rceil$ pages. The number of pages accessed by each task is shown in column 6 of Table I. The loop bounds of the *repeated access phase* is varied to select an appropriate execution time for a task. We use the *bubble sort, insertion sort, nth largest and statistics* benchmarks for our experiments. These benchmarks represent basic functionalities of a real-time task. Furthermore, we are constrained to small benchmarks since we only control DTLB entries that are not subject to OS noise under Linux.

# tasks	Task	Phase	Period	Execution	Pages
2	T1	1	2	0.4	16
	T2	0	54	27	16
3	T1	3	2	0.4	4
	T2	2	3	0.9	7
	T3	0	15	3.2	2
4	T1	3	2	0.2	2
	T2	2	3	0.6	2
	T3	1	5	1.1	1
	T4	0	15	3.2	2

TABLE I: Task characteristics

The experiments are designed in a way to show that our technique is applicable to a wide variety of scenarios. For two tasks, the periods are harmonic and the execution times are a combination of long and short runs. The number of pages accessed by each task is equal. The lower priority task T2 has more than one preemption. For three tasks, the periods are non-harmonic and tasks are simultaneously released. The number of pages accessed by each task differs.

Figures 14-17 show the average number of DTLB misses (y-axis) for a given task (caption) in a given task set (number of tasks per set on x-axis) across all jobs except the 0th (warm-up) job. *malloc_avg* and *tlb_malloc_avg* are the average results over five hyperperiods computed over five runs when the tasks use *malloc* and *tlb_malloc*, respectively, to allocate their pages. For the following graphs, *tlb_malloc_avg* results in zero misses. The graphs start at a negative value to make the value of *tlb_malloc_avg* visible.

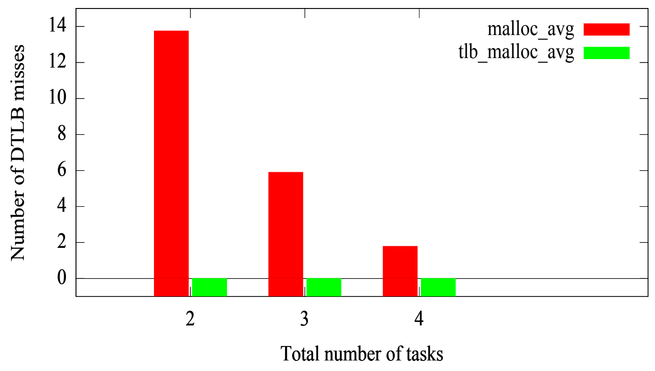

Fig. 14: T1

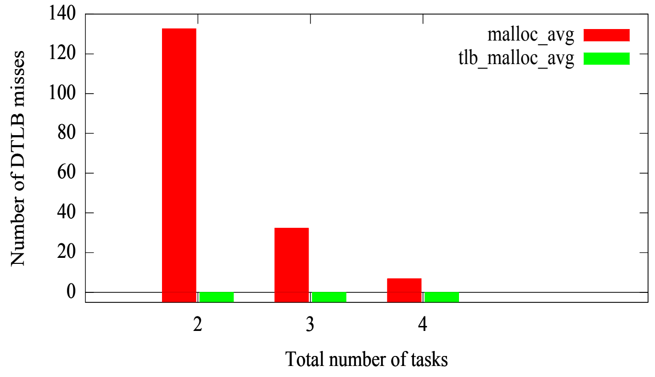

Fig. 15: T2

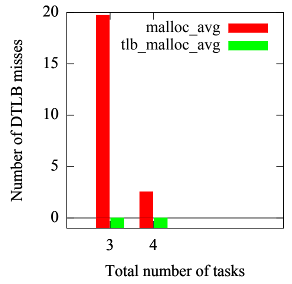

Fig. 16: T3

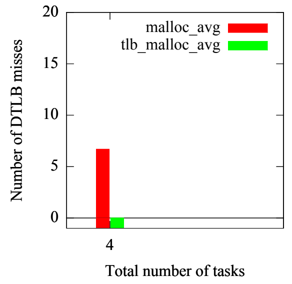

Fig. 17: T4

For the experiment with 2 tasks (Figures 14-15), if the tasks use *malloc*, we observe the number of DTLB misses to be about 13 and 130 for T1 and T2, respectively. If the tasks use *tlb_malloc*, both T1 and T2 incur no misses at all. The standard deviation of measurements for T1 and T2 are about 1.47 and 2.7, respectively, when the tasks use *malloc*. The standard deviation is zero for the two tasks when they use *tlb_malloc*.

For the experiment with 3 tasks (Figures 14-16), we can see that if the tasks use *malloc*, the number of DTLB misses for T1, T2 and T3 is about 6, 32 and 20, respectively. But if the tasks use *tlb_malloc*, then all three tasks incur zero misses. The standard deviation of results for T1, T2 and T3 are about 1.1, 5.5 and 2.5, respectively, when the three tasks use *malloc* (and zero for *tlb_malloc*).

For the experiment with 4 tasks (Figures14-17), the trend is similar. If the tasks use *malloc*, the number of misses incurred is around 1.7, 6.8, 2.5 and 6.7 on average for T1, T2, T3 and T4, respectively. If the four tasks use *tlb_malloc*, then each task incurs zero misses. The standard deviation of results for T1, T2, T3 and T4 are about 0.5, 0.8, 1 and 4.8, respectively, when the tasks use *malloc* (and zero for *tlb_malloc*). When the number of tasks is increased beyond four, the task set could not be isolated from OS noise resulting in DTLB misses even for *tlb_malloc*. This is because we use a stock Linux system instead of a real-time kernel and lack control for page allocation for stack and global variables.

The *adpcm and fft* benchmarks represent larger and more realistic workloads. We create two tasks with the following characteristics: T1 (1 ms, 4 ms, 2 ms) and T2 (0 ms, 20 ms, 4.6 ms) represent the *fft* and the *adpcm* benchmarks, respectively. The sum of the number of pages accessed by both tasks is twenty. Figure 18 shows the results of this experiment. The description of the graph is identical to the ones described above except that the x-axis represents tasks in this figure.

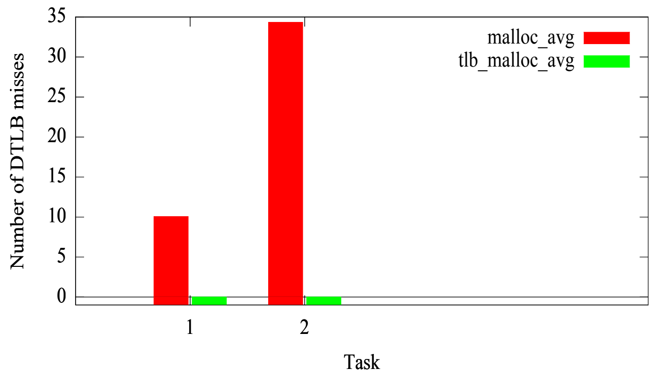

Fig. 18: T1 (fft) and T2 (adpcm)

In Figure 18, T1 and T2 incur about 10 and 34 DTLB misses, respectively, when the tasks use *malloc* (and zero for *tlb_malloc*). The standard deviation of the results for T1 and T2 are about 3.1 and 1.8, respectively, when the tasks use *malloc* (zero for *tlb_malloc*).

Figure 19 depicts the number of DTLB misses (y-axis) for task sets of 5-10 tasks of the Malardalen benchmarks statistics, nth-largest, bubblesort, insertion sort, and count (in the order of task indices, wrapping around after five tasks). Task T_i has has a period of 2^i milliseconds and increasing phases ($10-i$ms, except $5-i$ for 5 tasks and $6-i$ for 6 tasks). Each task colors a single page, and the number of heap pages is no more than four. Results are averaged over five runs of five hyperperiods each, as before. Each task set is connected by a solid/dashed line between each task (x-axis) for *tlb_malloc* and *malloc* (Buddy), respectively, also indicating the number of tasks in a set in the legend. The lower half of the figure is magnified for readability. Overall, the TLB-allocated tasks (solid lines) experience 50% fewer DTLB misses with

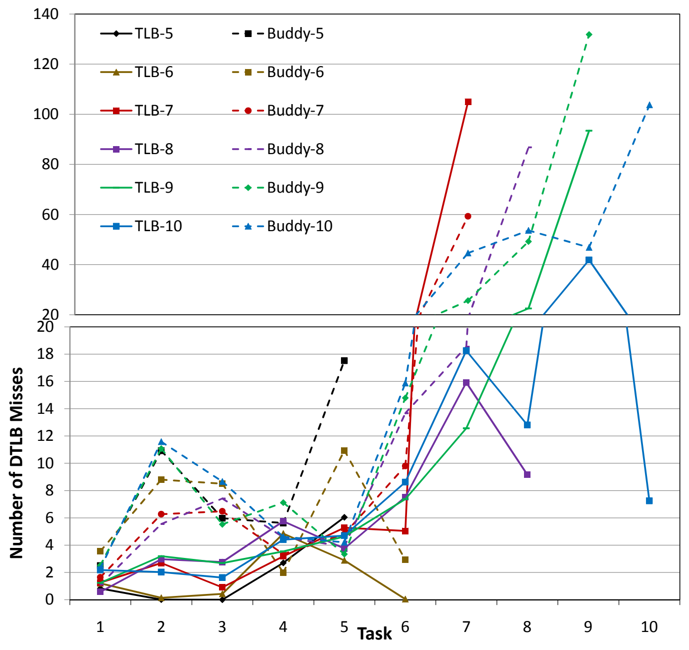

Fig. 19: DTLB misses for Set of 5-10 Tasks

a significantly lower standard deviation (less variability) than the buddy allocator (dashed lines). For most tasks, the number of DTLB misses is much lower with our allocator, but there are also rare exceptions where buddy has fewer DTLB misses (e.g., tasks 4+6 of Buddy-6, task 7 of Buddy-7). Furthermore, the number of DTLB misses increases for higher numbered tasks, and even tasks 1-3 per task set show up to 3 misses. This is caused by allocations of pages to DTLB cache sets, which is subject to OS noise. With larger task sets, we can no longer identify a sufficient number of non-noisy sets. For a commodity OS like Linux, only a small number of hard real-time tasks can provide full isolation, which underlines the need for a real-time OS and the need to incorporate coloring across all memory sections, not just the heap.

F. MiBench Suite

In this section, we describe experiments conducted with a mix of codes from the MiBench [9] and Malardalen suites to study a task set with longer execution times. We create two tasks, T1 (1 ms, 7 ms, 2.5 ms) and T2 (0 ms, 14 ms, 5.3 ms). T1 represents the *FIR* benchmark from Malardalen suite, while T2 represents the *adpcm* benchmark from MiBench. T1 and T2 operate on input data of 16 kB and 450 kB, respectively. The *adpcm* benchmark from the Mi suite operates on actual audio files and performs file IO, while the *adpcm* benchmark from the Malardalen suite described in the previous experiment operates on randomly generated data. Figure 20 depicts the results for this experiment. The description of the graphs is identical to that of Figure 18.

In Figure 20, T1 and T2 incur about 9 and 3 misses, respectively, when the tasks use *malloc* (and none for *tlb_malloc*). The standard deviation of the results for T1 and T2 are about 2.03 and 0.5, respectively, when the two tasks use *malloc* (zero for *tlb_malloc*).

Overall, we conclude that a set of tasks can be isolated from each other with respect to the DTLB if they use *tlb_malloc* to

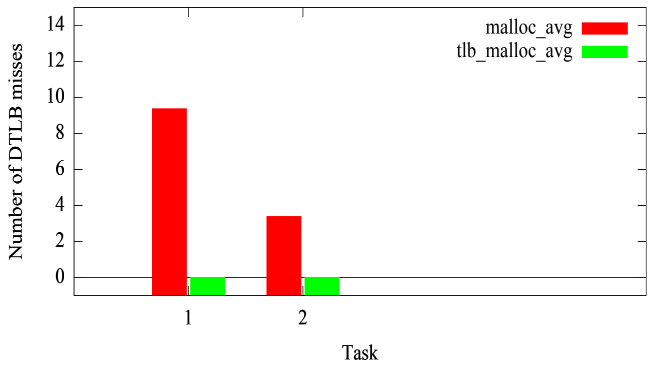

Fig. 20: Combination of Malardalen and Mi benchmarks

allocate their memory, not just for synthetic loads but for real-time codes as well. Nonetheless, such isolation is limited to few tasks for a general-purpose OS. Only a real-time OS with DTLB coloring across all data could extend this isolation to larger task sets.

VII. Discussion

TLB coloring is a software partitioning scheme and bears similarities with cache partitioning, including size considerations of partitions: TLB coloring is limited to the number of available sets (see Table 1), i.e., the number of preemptive real-time tasks should not exceed the number of sets. However, it would be feasible to reuse the same color for tasks with the same period, which are common in AUTOSAR environments. In that case, the cost of a single invalidation per colored TLB entry should be considered per task activation (in addition to the scenario of exclusive colors covered in our study), but interference due to preemption are still avoided.

Our implementation of TLB coloring is constrained to heap memory, but the coloring principle extends to stack and global data segments as well. The challenge here is to provide coloring within the kernel (i.e., the mmap system call for Linux), which we are currently working on. A remaining challenge is posed by treating the kernel (interrupt vectors, data, stack, heap) itself as a real-time task, which is only feasible for micro-kernels, and to ensure that TLB entries for hierarchical page tables itself are treated as kernel data during coloring.

The multicores featured here have private DTLBs while other architectures have shared L2 DTLBs [11], and research on a shared last-level TLB, albeit purely academic, exists [5]. Our approach could be extended to shared DTLBs as follows. Colors of tasks accessing common data (due to thread parallelism) could receive the same color while disjoint data would need to receive disjoint colors as well (across all cores).

VIII. Conclusion and Future Work

We presented the concept of *TLB coloring* in which virtual pages are colored such that pages of different color do not map to the same TLB set. Using this TLB software partitioning concept, we designed and implemented a heap allocator, *tlb_malloc*, to guarantee that a certain DTLB set holds a particular page translation.

We conducted experiments on the X86_64 architecture using a set of synthetic and standard benchmarks on a single core. Our finding is that *tlb_malloc* provides task isolation for real-time tasks under preemptive scheduling. Further, the number of DTLB misses are predictable using our allocator in the absence of OS noise (under perfect isolation). This combination of predictability and isolation should enable static timing analysis tools to compute significantly tighter bounds on the WCET and obviates the need to model TLB-related preemption delays.

In this work, we have demonstrated the feasibility of *TLB coloring* to provide inter-task isolation within the L1 DTLB. In the future, we plan to extend this idea to multi-level TLBs. We also plan to incorporate techniques of TLSF [16], CAMA [10] and PALLOC [26] to go beyond the capabilities of Suzuki et al. [22]. Incorporating these techniques will enable constant time dynamic memory allocation and provide task isolation with respect to TLBs, last level caches and DRAMs.

References

[1] Neal Altman and Nelson Weiderman. Timing variation in dual loop benchmark. *ACM SIGAda Ada Letters*, 8(3):98–106, 1988.

[2] AMD AMD. Bios and kernel developers guide (BKDG) for AMD family 15h models 00h-0fh processors.

[3] ARM ARM. Arm A-9 technical reference manual.

[4] MD Bennett and Neil C Audsley. Predictable and efficient virtual addressing for safety-critical real-time systems. In *Real-Time Systems, 13th Euromicro Conference on, 2001.*, pages 183–190. IEEE, 2001.

[5] A. Bhattacharjee, D. Lustig, and M. Martonosi. Shared last-level TLBs for chip multiprocessors. In *High Performance Computer Architecture*, pages 62–63, Feb 2011.

[6] Edouard Bugnion, Jennifer M Anderson, Todd C Mowry, Mendel Rosenblum, and Monica S Lam. Compiler-directed page coloring for multiprocessors. *ACM SIGPLAN Notices*, 31(9):244–255, 1996.

[7] Jack Dongarra, Shirley Moore, Philip Mucci, Keith Seymour, and Haihang You. Accurate cache and TLB characterization using hardware counters. In *Computational Science-ICCS 2004*, pages 432–439. Springer, 2004.

[8] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Languages, Compilers, and Tools for Embedded Systems*, pages 16–30. Springer, 1998.

[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workshop on Workload Characterization*, pages 3–14, December 2001. http://www.eecs.umich.edu/mibench.

[10] Jörg Herter, Peter Backes, Florian Haupenthal, and Jan Reineke. Cama: A predictable cache-aware memory allocator. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 23–32. IEEE, 2011.

[11] Intel Intel. Ia-64 architectures software developers manual. *Volume 3A: System Programming Guide, Part*, 1.

[12] Takuya Ishikawa, Toshikazu Kato, Shinya Honda, and Hiroaki Takada. Investigation and improvement on the impact of TLB misses in real-time systems. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 5–10, July 2013.

[13] Jochen Liedtke, Hermann Hartig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 213–224. IEEE, 1997.

[14] Malardalen. WCET benchmarks. Available from http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[15] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54. IEEE, 2013.

[16] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 79–88. IEEE, 2004.

[17] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2-3):217–247, 2000.

[18] Isabelle Puaut and Damien Hardy. Predictable paging in real-time systems: A compiler approach. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 169–178. IEEE, 2007.

[19] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *ACM Transactions on Embedded Computing Systems*, 10(2):DOI 10.1145/1880050.1880063, December 2010.

[20] Harini Ramaprasad and Frank Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 148–157. IEEE, 2005.

[21] Harini Ramaprasad and Frank Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 71–80. IEEE, 2006.

[22] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *International Conference on Computational Science and Engineering*, pages 685–692, 2013.

[23] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 157–167. IEEE, 2013.

[24] Randall T White, Frank Mueller, Christopher A Healy, David B Whalley, and Marion G Harmon. Timing analysis for data caches and set-associative caches. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 192–202. IEEE, 1997.

[25] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problemoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[26] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), to appear*, 2014.