Increasing Memory Utilization with Transient Memory Scheduling*

Qi Wang*, Jiguo Song*, Gabriel Parmer*, Andrew Sweeney†, Guru Venkataramani*

* The George Washington University
Washington, DC
{interwq,jiguos,gparmer, guruv}@gwu.edu

† AppNexus
New York, NY
asweeney@appnexus.com

***Abstract*—In addition to predictability, both reliability and security are increasingly important for embedded systems. To limit the scope of errant behavior in open and mixed criticality systems, a common approach is to raise isolation barriers between software components. However, this decentralizes memory management across all system components. Memory is often cached and quickly accessible in each application.**

This paper introduces the TMEM system for increasing memory utilization while optimizing for application end-to-end constraints such as meeting deadlines. In addition to the traditional spatial multiplexing of memory, TMEM introduces the predictable temporal multiplexing of memory within caches in a system component, and *memory scheduling* to continually reallocate memory between components to best benefit the system. We find that TMEM is able to maintain the efficiency of caches, while also lowering both task tardiness and system memory requirements.

I. INTRODUCTION

Embedded and real-time systems are increasingly required to provide not only predictability, but also increased reliability, security, and isolation guarantees. Open real-time systems in which hard real-time tasks execute along-side best effort and untrusted applications require not only that the real-time tasks meet their deadlines, but that they are isolated from the possibly faulty or malicious programs. This motivates a class of systems that provide fault-isolation at a finer granularity than is typically provided by monolithic operating systems and applications. Hardware techniques for memory isolation (*e.g.* page tables) are commonly used to segregate the functionality of the system into separate *components*. A fault in one component cannot access or corrupt the memory in another (e.g. if hardware constraints components to accessing disjoint sets of physical memory), thus constraining the fault propagation and the adverse effects of buggy or malicious components. Examples of such systems include μ-kernels [1], component-based OSes [2], [3], and middle-ware systems [4]. Here we use "components" to broadly encapsulate all of these techniques for decomposing the system. Section IV-A will refine these definitions for our implementation.

The separation of memory into isolated components complicates the memory management and allocation of the system. Due to the high costs of mapping in and unmapping memory from components, it is common for memory to be allocated to components that in turn use libraries to treat the memory as a cache for finer grained memory allocations. The obvious example is the implementation of `malloc` and `free` that uses kernel `sbrk` or `mmap` system calls to request memory that is then managed using data-structures defined by the specific `malloc` implementation. Memory management libraries often cache unused memory to speed up future allocations. We denote the amount of memory in use and being requested at any point by threads in component c^j, r^j, and the amount of memory allocated to specific components by the system, a^j. Due to the caching of allocated memory, it is common for $\sum_{\forall i} r^j < \sum_{\forall i} a^j$, or for the system to have low *memory utilization*.

The distributed management of memory across components leads to undesirable resource allocation properties. For example, a real-time thread that requires memory in c^j might be unable to satisfy that request even though memory is unused in other components ($\exists i \neq j | r^i < a^i$). For this reason, it is common to pre-allocate and *pin* memory in real-time components. This ensures that memory is available when requested in that component, but conservative preallocation decreases average-case memory utilization ($r^j \ll a^j$), thus requiring more system memory. This negatively impacts system cost, size, and power usage, all limiting factors in embedded systems.

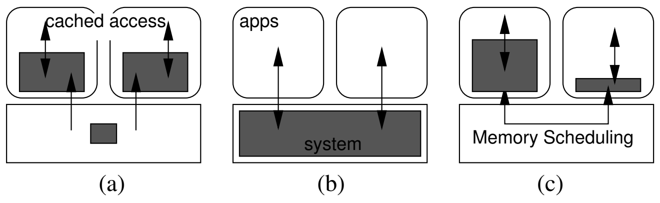


Fig. 1. System organizations for managing memory – memory caches are grey. (a) Conventional OSes manage memory as decentralized heaps that cache memory allocated from the kernel. (b) Centralized memory management with expensive allocation and deallocation. (c) TMEM with user-level caches for fast memory access with predictable resource sharing within the cache, and *memory scheduling* to move memory between caches to optimize for thread end-to-end constraints.

Figure 1(a) depicts the traditional system design with memory management distributed across system components. Figure 1(b) depicts an alternative system design in which all allocations and deallocations are requested from a centralized manager. Allocations from the manager will be on a page granularity and when a page is unused, it is returned immediately to the centralized memory manager. Importantly, when under memory pressure, this enables the centralized and intelligent allocation of memory across the entire system.

This material is based upon work supported by the National Science Foundation under Grants No. CNS 1137973, CNS 1149675, and CNS 1117243. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

However, the cost of mapping and unmapping pages from the page-tables of each component is significant. Figure 1(c) presents TMEM. Component caches provide fast, predictable access to memory within that cache, and memory scheduling is used to periodically move memory between components to optimize for thread end-to-end constraints. TMEM is a dynamic approach that adapts to observed deadline misses. Therefore it is not appropriate for hard real-time systems.

Transient memory. This paper details the TMEM system to achieve the benefits of both structures in Figure 1: low worst-case overhead when memory is available, and with bounded overheads when it is not to provide high system-level memory utilization. In this paper, we focus on a specific type of memory we call *transient memory*. Transient memory is defined by the following properties:

P1. The memory is dynamically allocated.

P2. The interval between memory allocation and deallocation is bounded. Functions with a bounded execution time that use memory following a scoped allocation lifetime (*i.e.* it is deallocated before leaving a lexical scope), naturally satisfy this property. In real-time systems where the worst-case execution time (WCET) of code in a component is known, the WCET naturally places the bound on TMEM hold time.

P3. The maximum allocation size is bounded a-priori. This enables TMEM to ascertain how much memory will be guaranteed to satisfy a single allocation.

P2 is the largest divergence from traditional memory management schemes, and gives transient memory its name. The bounded lifetime property enables it to be treated as a traditional shared resource, thus temporally multiplexing it between different threads. For example, there might not be enough memory in the system to concurrently satisfy two thread's requests. However, it is possible to allow one thread to access the memory which will, within a bounded time, then be deallocated. The system will then satisfy the other thread's request. We investigate how to apply the Priority Inheritance Protocol (PIP) [5] to bound allocation time.

Transient memory enables temporal multiplexing of memory in addition to conventional spatial multiplexing, thus introducing a form of *CPU/memory co-scheduling*. TMEM uses both forms of resource multiplexing to optimize for thread end-to-end constraints (*e.g.* by dynamically reducing task tardiness when it is observed). TMEM uses spatial multiplexing (as in Figure 1(a)) to maintain lower task execution times. In low-memory situations, TMEM mimics Figure 1(b) and temporally multiplexes memory which maintains predictability, but increases response times. The system uses *memory scheduling* to allocate memory to components over windows of time to explicitly optimize for thread end-to-end constraints.

To demonstrate the TMEM subsystem, we focus on the implementation and evaluation of two *types* of transient memory in the COMPOSITE component-based OS [2]:

- *IPC Execution stacks.* When a server component is invoked by another component (*i.e.* a service is requested via Inter-Process Communication (IPC)), the invocation must execute using an execution stack (C code assumes a stack). The memory for this stack is only needed for the extent of the invocation. If multiple requests are made concurrently for the same component, and the concurrency level outstrips the amount of local memory, contention management on the stacks is required.
- *Efficient shared memory.* IPC between components is often modeled as synchronous function calls and requires some means to transfer data between components. Shared memory is often used to avoid inefficient data copying. The memory is required for the scope of the IPC, and often not after (until the next invocation). As with stacks if a component's concurrency level outpaces the amount of local shared memory available, contention management on the shared buffers is required.

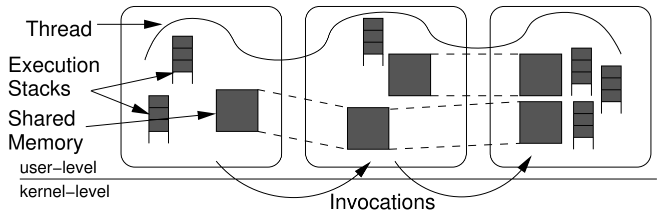


Fig. 2. Transient memory in COMPOSITE. IPC is conducted in COMPOSITE using *thread migration* [6] between separate user-level components. The same schedulable entity migrates between components via invocation. Components are in separate memory protection domains, thus invocations require kernel mediation. Separate C *execution stacks* must be used for the thread's execution in each component, and shared memory is allocated to pass data. Both forms of transient memory (stacks and shared memory) are depicted in dark grey.

Execution stacks and shared memory in COMPOSITE are depicted in Figure 2.

Importantly, the amount of these types of memory requested in a component is *a function of the concurrency* in that component. If only a single thread ever invokes a component, there is no need for more than one stack. However, if ten threads are invoking a component, it would request *up to* ten stacks (if the invocations were sequential, and not concurrent, then again, only a single stack is required). It is difficult to predict the concurrency of all components in a system, especially in an open system.

Example. If many threads wish to send data onto a network, in a system with pervasive fault isolation, a number of components including UDP, IP, and the networking device would be involved. Before any packets are sent, shared memory is allocated for the thread, and passed through all these components. When the network processing is complete, the allocation in the shared memory is no longer required. When IPC invocations are made, a stack is used for execution in the called component's protection domain. When the processing is complete, the stack is available for use for another invocation. The bounds on the lifetime of memory allocations match the lifetime of the function invocations (which is bounded in a real-time system), thus satisfying P2.

More general memory management functions such as scoped memory management [7] also follow the transient memory allocation/deallocation pattern. Though we focus on these two forms of transient memory, we discuss the generic system interfaces that enable the addition of other types of transient memory.

Contributions. This paper makes the following contributions. (1) We introduce the scheduling of transient memory as a novel and useful means for increasing the memory utilization of the system, while still meeting the end-to-end constraints of individual threads. We do not know of previous work to provide efficient, predictable access to memory while optimizing for thread end-to-end constraints. (2) We study two forms of transient memory that represent novel mechanisms for managing state in systems with fine-grained fault isolation and frequent IPC. (3) We detail the design and implementation of transient memory management and scheduling in the COMPOSITE component-based OS. (4) We describe a scheduling algorithm for mapping memory to components to optimize for both high memory utilization, and the removal of tardiness when it is observed. (5) We evaluate the proposed system and find that it can maintain consistently low task tardiness, and adapt memory allocations in open real-time systems to the unpredictable execution of best-effort threads.

This paper is organized as follows. Section II discusses related work. Section III discusses the organization and goals of the TMEM subsystem for transient memory management. Section IV covers the implementation of the two different types of TMEM in COMPOSITE, and how they use the TMEM mechanisms. Section V details the scheduling policy used to split memory between components over each period of time. We evaluate the transient memory infrastructure in Section VI. Section VII discusses conclusions.

II. RELATED WORK

Conventional approaches for coping with memory shortages. Virtual memory is commonly used to extend memory by swapping out less frequently accessed pages to disk. Due to the unpredictable delays this can induce on memory accesses, virtual memory is often disabled (by pinning memory) for real-time processes. Preventing paging for soft real-time processes requires conservative memory allocation, thus trading predictability for memory utilization. Further, many embedded systems do not have large enough storage devices to support full virtual memory. Systems that experience memory pressure that cannot support virtual memory either 1) return error codes for memory allocations (*i.e.* NULL from `malloc`), or 2) abruptly terminate system processes. TMEM is significantly less general as it does not support arbitrary forms of memory. However, TMEM ensures the predictable access to memory via temporal multiplexing of memory within a component cache, and intelligent memory scheduling between components to attempt to remove any observed tardiness in the system.

Virtual memory and swapping. Redline [8] and both [9] and [10] take a full-system approach toward the dynamic resource management of CPU and memory in a commodity system. They use swapping to increase effective memory capacity, and allocate resources to meet application constraints. TMEM takes a fundamentally different approach that makes specific trade-offs.

- TMEM doesn't require a disk to swap inactive memory to. This is appealing for smaller embedded systems without such access.
- TMEM memory allocations will always be fulfilled within a bounded amount of time. When cached memory is available, the bounds are less than if temporal multiplexing is required. However, even in that case, allocation times are bounded, though possibly large due to resource sharing protocol overheads. In fact, TMEM could be used for hard real-time systems that assume a worst-case memory allocation overhead of contended access, but in such cases there is little benefit to optimizing the uncontended, cached case. In contrast to TMEM, it can be difficult to manage a disk and swap predictably and put a reasonable bound on memory access time.
- TMEM works only on a restricted subset of memory whereas swapping works on all user-memory (and in some systems, for some kernel memory as well).

Given the trade-off between the techniques, it is possible that both techniques could be used collaboratively to achieve some of the benefits of both.

System shared memory. Zero-copy data movement is essential for system efficiency, especially in systems decomposed into separate protection domains. Techniques such as Fbufs [11] have been proposed to use persistent shared mappings to move data. IOLite [12] extended this system to include all system buffers. CBUFS, or COMPOSITE buffers – our shared memory technique – use similar techniques to FBufs to map buffers across components, but integrates these facilities with the ability of TMEM to remove and remap buffers to separate components. To avoid overhead, remapping is done infrequently, and in the common case, cached mappings are used. Miller et al. introduced methods for minimally copying data streaming between interfaces [13], and RAD-FLOWS [14] defines how large a buffer must be to store data from asynchronous communication between real-time threads. These approaches are complementary to TMEM and can be used to determine hard reservations for worst-case buffer size. TMEM focuses on a dynamic approach of memory scheduling that makes it less suitable for hard real-time.

Execution context sharing. When a server component provides services to multiple clients, that server must allocate resources to handle each request. These resources can take the form of specific threads that are used to handle client requests in middleware [4] and μ-kernels, or as memory to be used as a C execution stack as in COMPOSITE [15]. The size of the pool of these resources has an effect on the predictability of the system, and the best size is a function of the concurrency and nature of the requests. In [15], we compared different resource sharing protocols to achieve system schedulability with a *static allocation* of stacks to specific component caches.

In this paper, TMEM does not focus on hard real-time systems, and instead adapts memory allocations of both stacks and shared memory to dynamically remove task tardiness when it is observed.

Predictable, efficient inter-process communication. Blackham et al. [16], [17] have researched the overheads of the kernel interface, and of interrupt latency in a system that provides component-based protection. TMEM focuses on predictable access to memory resources used for communication between components. Given a predictable kernel, this work provides a foundation for predictable inter-component communication (via RPC) while controlling memory usage. Specifically, invocations in COMPOSITE are predictable in that they are bounded if the execution time within the called function is bounded, and both priorities and reservations are automatically inherited (via thread migration [6]). This work provides the mechanisms for controlling interference from lower-priority threads in system components.

Predictable multi-unit resource sharing. When a thread requests TMEM, a predictable resource sharing protocol must arbitrate access to TMEM in a component's cache. We use PIP [5] as detailed in Section IV-B. Both the Stack Resource Protocol (SRP) [18] and the Priority Ceiling Protocol (PCP) [19] also provide predictable resource sharing. We show in [15] how these protocols can be used in a schedulability analysis. This work shows that in COMPOSITE no one protocol dominates the others for all system configurations. In this paper, we choose PIP and focus on the main contributions of TMEM in this paper. In some systems, SRP or PCP might be more beneficial as PIP suffers from adverse scheduling effects in the presence of nested resources. Regardless of the protocol used, we show in Section VI that TMEM can be used to control tardiness even in the absence of a predictable sharing protocol. However, in such a case, each individual invocation is no longer bounded.

Transcendental Memory. Another technology using the name TMEM (for Transcendent Memory [20]) is used in Virtual Machines (VMs) to manage cached memory and avoid swapping VM memory out to disk. This work focuses on increasing system memory utilization to maximize performance (*i.e.* avoid disk I/O). In contrast, this paper focuses on the predictable sharing of memory within component caches for predictable memory access, and memory scheduling to resize the caches to optimize for task end-to-end constraints.

III. DESIGN OF TRANSIENT MEMORY MANAGEMENT

The TMEM subsystem for managing transient memory has the following high-level goals related to obtaining the benefits of both Figure 1(a) and (b):

G1 *Per-component caching.* When $r^i \leq a^i$, TMEM must enable efficient access to per-component cached transient memory. If requests for transient memory can be guaranteed to be satisfied from within the cache, the worst-case allocation time is low due to spatial multiplexing.

G2 *Predictable cached transient memory contention.* When $r^i > a^i$, the concurrency of the component is greater than the amount of memory in the component's cache. TMEM must either add spare memory into the cache, or block the thread until memory is available, thus temporally multiplexing the memory. Importantly, the *latency between the request, and the eventual allocation must be bounded.* We refer to this as *predictable* access to memory in the cache. This bounded property is due to a combination of the use of a predictable resource sharing protocol (*i.e.* priority inheritance, PI), and property P2.

G3 *Transient memory scheduling toward end-to-end application goals.* When the $\sum_{\forall c^i} r^i > \sum_{\forall c^i} a^i$, TMEM must partition the available memory between components, thus forcing some contention and overhead on memory in caches in some components. Over time, as workloads change, the optimal allocation to components will change to meet their end-to-end constraints. This memory scheduling requires a sub-goal: **G3a.** TMEM must be able to remove transient memory from a component asynchronously, and without the component's involvement so that it can be moved to another component to decrease key thread's end-to-end latencies.

The TMEM system design focuses on (i) maintaining efficient and predictable access to memory *within a component's cache* (G1 and G2), and (ii) memory scheduling (G3) is used relatively rarely to rebalance memory allocations between component caches. This maintains the benefits of Figure 1(a) and (b), while providing predictable memory access within a component. We make this separation between the "fast (predictable) path", and the memory scheduling because moving a set of pages from one component to another is expensive due to clearing the memory and page-table manipulations (requiring TLB flushes). This design achieves the benefits of having low-cost common-case allocations, predictable worst-case allocation times, and the ability to balance memory between components over time to adapt to thread end-to-end constraints.

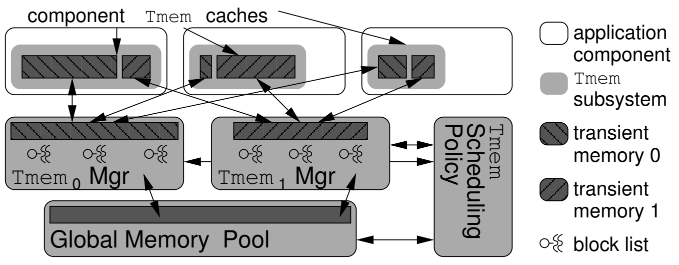


Fig. 3. The TMEM system organization. Applications cache different types of transient memory, which is allocated by different TMEM managers. TMEM managers block application threads when more memory is requested than can be handled by the cache. The managers are allocated memory out of the global pool of memory, and the policy dictates how much memory to allocate to each component.

Figure 3 shows the TMEM system organization. In a monolithic system, a reasonable separation would place all of TMEM in the kernel aside from the caches. Due to the focus on fault isolation, COMPOSITE implements each as separate, hardware-isolated components.

IV. TMEM IMPLEMENTATION

A. COMPOSITE *Background*

We use the COMPOSITE [2] component-based OS as a test-bed to study TMEM. COMPOSITE focuses on decomposing system software into hardware-isolated, user-level components. Components include the code and data associated with some functionality, and export an interface of functions that can be invoked by other components. In COMPOSITE, even low-level system services such as scheduling [21], synchronization [6], physical memory management, I/O handling, and event management are defined by replaceable components. This simple model enables complicated policies such as hierarchical resource management [22]. Each component is encapsulated and opaque behind this interface, enabling them to be separated into different protection domains, provided by hardware mechanisms (page-tables). A functional system is the composition of a set of components. Each component has caches of memory that are managed and sized by the TMEM subsystem. Threads are the active entities in the system and are executing at any point in time in a specific component. To begin execution in another component, an invocation is made on a function in its interface.

Component invocations. Both forms of transient memory are used pervasively in COMPOSITE, and their management is required for the overall system to be both efficient and predictable. *Component invocations* are used to refer to synchronous IPC between components as the main form of communication between components is semantically identical to function invocation. COMPOSITE uses thread migration [6] for IPC: the same schedulable entity executes across many components, and simply uses different execution contexts in each component.

All arguments for an invocation are passed in registers. To keep the kernel invocation path simple and efficient, the kernel does not copy or map memory. If a function requires more arguments than there are registers (*i.e.* there are only 6 general purpose registers on x86-32), or if the arguments are pointers to data regions, an Interface Definition Language (IDL) compiler generates stubs to pass the data in shared memory using CBUFS. Thus, for the typical programmer, the use of CBUFS and the management of execution stacks is completely transparent.

Despite the decomposition of system software into hardware-separated components which requires extensive IPC, the efficiency of COMPOSITE is reasonable. In [23], we show that a web-server implemented as more than 25 components (causing over 70 component invocations per HTTP client request) is competitive with traditional software (apache on Linux). The COMPOSITE kernel is less than 6K lines of code (compared to millions of lines of code for monolithic systems such as Linux). All component code, including 3rd party libraries, totals over 100K lines.

B. *Component-Local Transient Memory Management*

There is a TMEM manager for each type of transient memory (in the paper, stacks and shared memory). Thus we describe a stack manager, and a CBUF manager (CBUFS are COMPOSITE shared-memory buffers). All TMEM managers interact with the TMEM caches to (i) add memory to the cache, (ii) remove memory from the cache (iii) and predictably mediate contention on the memory in the cache. Toward this, all TMEM managers and client cache code implement protocols for the allocation and deallocation of memory. These protocols involve an API the client calls when no memory is available in the cache, or when the manager wants to reclaim memory, and shared memory structures mapped into both the client and the manager to track TMEM. The latter are used by the manager to asynchronously remove unused transient memory from the client (*i.e.* when it is required elsewhere).

Data-structures shared between managers and per-component caches. The shared structures include:

- A *relinquish bit* is shared that is marked by the manager when it wants the client to give up some memory, but it is all currently allocated. This is used (i) to signal that memory is required by a higher priority thread that is blocked waiting for it (for G2), and (ii) if the manager wishes to shrink the allocation of memory to the component (for G3). When a thread in the client deallocates TMEM, it checks this bit, and calls the manager if it is set to release the memory.
- TMEM index tables that are used to identify and retrieve pieces of transient memory. These indices include (i) information about transient memory addresses, and (ii) the thread id of the thread using the transient memory – used to perform priority inheritance (G2).

TMEM manager API. The main functions exported by the TMEM managers include:

- `tmem_get` – When a TMEM allocation cannot be satisfied by the client's cache, this function is invoked. The manager will set the relinquish bit in the client so the manager will be notified of any TMEM deallocations within that component. Then the manager performs predictable resource sharing on the TMEM resources within the client. Though the system could use any predictable protocol including multi-unit SRP [18] or PCP [19], our implementation uses a simple PIP. The thread requesting TMEM in the manager will determine one of the threads holding TMEM in the cache of client (chosen arbitrarily for simplicity). That thread receives the priority of the higher-priority contending task. This implementation bounds the interference from the lower-priority resource holding task to a single access (G2) in a manner equivalent to single-unit PIP[1].

[1]COMPOSITE uses a mechanism similar to shadow tasks in Shark [24] to implement priority inheritance: instead of removing a contending thread from the runqueue, a dependency is simply annotated in the thread structure. When a scheduling decision is made, if the task has a dependency, then the dependency target is dispatched instead. The dependency chain is bounded by the maximum nesting of resources. A more traditional PI (or PCP) implementation could be used instead.

- `tmem_put` – When a thread deallocates TMEM, and the relinquish bit is set, it calls this function to inform the manager to wake the higher-priority thread waiting for TMEM. The manager will unset the relinquish bit if no threads are left blocked waiting for memory, thus preventing threads that release TMEM from unnecessarily calling the manager.

Component TMEM usage meta-data. Each manager maintains meta-data about a component's (c^j) transient memory usage that will be used by the memory scheduling policy: (i) The policy needs to understand how much memory is concurrently requested, r^j. When threads call `tmem_get`, this is updated to reflect a^j plus the amount requested by threads blocked waiting for an additional allocation. Additionally, whenever TMEM is accessed in a component's cache a *used* bit is set in the TMEM index. To detect r^j values *less* than a^j, TMEM that is unused between successive runs of the policy is detected by observing unset used bits, thus lowering the r^j values. (ii) Each manager tracks b_i^j, the amount of time a thread τ_i spends blocked waiting for transient memory in component c^j, and the number of times it blocked. This enables the policy to understand which components contribute to an increased lateness (enabling G3).

C. TMEM *Execution Stack Management*

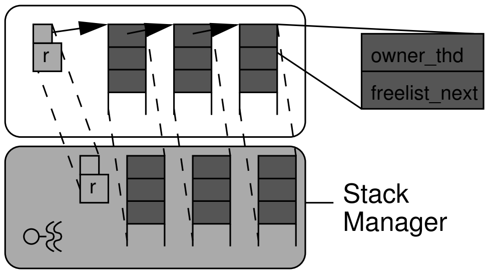


Fig. 4. The stack manager and application cache data-structures. In the application component, stacks are maintained in a freelist. When a thread takes a stack, it places its id at the stack top. The stack manager has access to the freelist head, the relinquish bit, and all stacks, thus is able to add or remove stacks asynchronously. Shades are coded as in Figure 3.

Figure 4 depicts the shared data-structures between the TMEM stack manager and each component, and the structure of the cache. The TMEM index table used to retrieve stacks is a simple freelist. This structure provides efficient caching toward G1.

Client stack management. When a thread upcalls into a component (*i.e.* to invoke a function in the component's interface), it first checks to see if a stack exists on the freelist ($r^j \leq a^j$). If so, it is removed from the list atomically. Otherwise ($r^j > a^j$), it calls `tmem_get` in the manager. Upon return from this component, it places the stack back on the freelist (to be reused in the future), and checks if the relinquish bit is set. If so, the stack manager's `tmem_put` is called. This simple protocol satisfies both G1 and G2.

Stack manager asynchronous removal of stacks from components. The stack manager component is able to add stacks to the component's freelist, and remove them, at will (G3a). COMPOSITE uses a hierarchical resource management strategy described in [22] in which memory in a component can be mapped into a "child", and removed in the future. The stack manager uses this support to map stacks into each component, and also to map in the initial freelist head. It can walk the freelist, and remove specific stacks by simply inspecting the memory it has mapped into the component (after translating addresses in the freelist into virtual addresses local to the stack manager). Accordingly, when the TMEM policy determines that stacks should be allocated into a component c^j, they can be asynchronously removed from another component and mapped into c^j.

D. TMEM *Shared Memory Management*

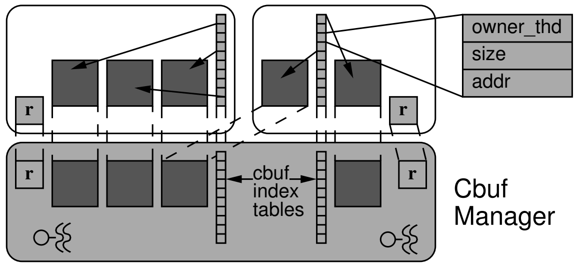


Fig. 5. The CBUF manager and application component data-structures. Memory is mapped into components, and can be shared between components (as in the 3rd mapping). Both the application and manager can access CBUFs through their `cbid` that indices into the CBUF table. Thus, CBUFs can be added and removed asynchronously by the manger. This diagram omits the slab metadata. Shades are coded as in Figure 3.

Efficient movement of data between components is essential for reasonable end-to-end performance. Shared memory is used to avoid copying data between protection domains. Due to the cost of constructing shared memory mappings, efficient shared memory systems [11] *cache* mappings and reuse them across multiple instances of IPC. CBUFs are designed with this in mind, and cache shared memory allocations in each component.

CBUFs currently assume that a component will both allocate and deallocate its buffers, and that the mappings into other components are valid for the scope of component invocations. Thus, shared memory follows the transient memory allocation pattern: a CBUF is allocated, used to transfer data to other components with an invocation, and when the invocation returns, the CBUF is deallocated. CBUFs could be extended to enable more general lifetimes to support caching [12], but we leave this as future work.

Operation	Description
`void * cbuf_alloc(size, *cbid)`	allocate a cbuf from the cache
`void cbuf_free(void*)`	deallocate a cbuf
`void * cbuf2buf(cbid, size)`	get a possibly cached mapping for `cbid`

TABLE I
MAIN CBUF OPERATIONS.

CBUF cache implementation. Each individual CBUF has a unique identifier, its `cbid`. As data can only be passed in a limited number of registers between components, CBUFs are designed such that only the `cbid` and size of the data need be passed, and in the invoked component, these can be converted into the appropriate shared memory mapping. Table I lists the operations components perform on CBUFs. These include allocation and deallocation (which will be done from the component's cache of transient memory), and `cbuf2buf` which is called in an invoked component and converts between the passed in `cbid` to the actual shared memory mapping. These mappings are cached across invocations to avoid the high cost of mapping and unmapping the memory for each invocation.

Figure 5 depicts the CBUF manager, and its shared data structures with components. In each component, the CBUF cache is managed using slab allocation [25] (toward G1). Slab meta-data is stored separately from the CBUF memory, to avoid its corruption by another component.

Two index tables are maintained in each component. (i) The TMEM index table associates `cbids` with the actual transient memory buffers in the component, and contain meta-data such as the owner thread id of a thread currently accessing the CBUF, and the size of allocations within the CBUF. This table is implemented using shared memory between each component, and the CBUF manager. (ii) A table is used to map the address of the buffer to the slab descriptor containing meta-data about it. When a buffer is freed, this enables the slab meta-data to be updated. Both of these tables are implemented as radix trees with a fixed depth of 2, thus enabling predictable and efficient lookups.

When `cbuf_alloc` is called, and there is no CBUFs available, then `tmem_get` is invoked to perform priority inheritance with a thread currently allocated a CBUF in the component. This is done by setting the relinquish bit in the component, obtaining the thread id from the first table (that is shared with the CBUF manager), and invoking `sched_block` with a dependency.

Whenever `cbuf_free` is called and all memory in a page is deallocated, then the relinquish bit for CBUFs is checked. If it is set, then `tmem_put` is invoked. This enables the CBUF transient memory manager to remove the memory from the component, and move it to another component as dictated by the transient memory scheduling policy.

`cbuf2buf` uses the TMEM index table in an invoked component to find the shared memory passed with an invocation. If the mapping is cached (*i.e.* it is in the table), then this operation is fast. However, if the component has not mapped in that `cbid`, then it must call the CBUF manager to map it in. The manager will add the mapping into the component, and into the TMEM index so that future `cbuf2bufs` will find the cached mapping.

CBUF manager asynchronous memory reclamation. To remove CBUF memory from components so that it can be allocated elsewhere according to the transient memory scheduling policy (G3a), the CBUF manager iterates through CBUFs within the component, finds any that are not in use, and removes them from the TMEM index tables of all components they have been mapped into before actually unmapping the memory.

E. System-Wide Transient Memory Management

When $\sum_{\forall c^j} r^j > \sum_{\forall c^j} a^j$, the TMEM scheduling policy must decide how large the TMEM allocations should be to each component. Essentially, this policy must minimize the end-to-end costs that blocking waiting for TMEM has on threads to best utilize system memory. It does so by setting a *desired transient memory allocation*, δ^j, for each component, c^j, in the TMEM manager. If the desired allocation to a component is less than its current allocation ($\delta^j < a^j$), then the manager first attempts to asynchronously remove memory from the component. If it cannot remove enough memory because threads are actively using it, the relinquish bit for that component is set so that currently used allocations will be returned when deallocated. In such a case, $a^j > \delta^j$ for a period of time bounded by the amount of time a thread keeps the memory (*i.e.* a bounded extent due to P2). Thus there can be a lag, in the worst case, between when the policy wishes to move TMEM to a different component, and when it is actually moved (*i.e.* until when $a^j = \delta^j$).

Removed pages are added into a global pool in the TMEM manager. If $\delta^j > a^j$, then when threads call `tmem_get`, the manager will pull pages out of the TMEM manager's pool and add them into the component (also adding them into the component's TMEM index).

If $\sum_{\forall c^j} \delta^j \neq \sum_{\forall c^j} a^j$ for a TMEM manager (stack or CBUF), then memory is either retrieved from or given to the global memory pool. This global memory pool holds all memory allocated to transient memory in the system, and it is partitioned between different TMEM managers by the policy by setting δ^j for each component.

Ensuring predictable progress. If a thread invokes a component with no transient memory and attempts an allocation, the thread will be blocked without the ability to perform predictable resource sharing with another thread (*i.e.* no other thread has TMEM in the component's cache). It will have to wait for the memory scheduling policy to move TMEM to the component. Instead of suffering this latency, P3 enables us to know the largest allocation required, and we preallocate this to each component.

Deadlock prevention. Deadlock is generally possible as threads might hold transient memory resources in one component, while waiting for them in another. If these components were arranged such that communication between them could be cyclic, deadlock could occur. To prevent this, we make the assumption that the synchronous invocation pattern between components is not cyclic. This is an assumption of COMPOSITE unrelated to TMEM, as it is difficult with invocation cycles between components to bound execution.

Multi-core support. TMEM currently only works on a uniprocessor. We envision a multi-core adaption of this system using TMEM pools partitioned across cores. Not only does this make management of the memory easier, but also it is important for performance to avoid cache migrations for memory shared across cores. Thread migrations might result in TMEM that was allocated on one core, being deallocated on another. Such inconsistencies would have to be dealt with by the managers (similar to how heap memory allocators often do local allocations, and balance between cores).

Security concerns. TMEM relies on clients following the specified protocols for interaction with the managers. If the clients do not do so, they might attempt to obtain, or retain more than their due share of memory. This is generally a problem in any system that relies on client behavior to drive allocation decisions. For example, in a virtual memory system, processes can frequently access all of their memory to give the kernel the impression that all the process's memory is "hot", and should not be swapped out. TMEM provides a large benefit here: due to P2, TMEM managers can track if memory hasn't been deallocated within the bounded amount of time. If it hasn't then either the process is maliciously attempting to avoid giving up the memory, or it is accidentally doing so. In either case, remedial action is required, perhaps restarting the client.

V. TRANSIENT MEMORY SCHEDULING

We separate the TMEM scheduling policy from the *mechanism* of each individual TMEM manager. The policy interacts with each manager through a well-defined interface that is used to gather information about thread performance, and to manipulate the amount of transient memory given to each component. Specifically, the policy wishes to determine the amount of time specific threads spend blocked contending a transient memory allocation (b_i^j), and how much those threads deviate from an execution target (*e.g.* meeting deadlines).

System model. To optimize for the end-to-end constraints of system threads, we use a simple system model. The system has a number of periodic tasks, $\tau_0, \ldots, \tau_n$, each of which is released at a periodicity p_i, and executes for a worst case execution time of e_i each period, thus its utilization is $u_i = e_i/p_i$. The deadline, $d_{i,j}$, of the jth release of task τ_i is its release time $+p_i$. The tardiness of a task is $t_{i,j} = \max(0, c_{i,j} - d_{i,j})$ where $c_{i,j}$ is the completion time of the jth release for τ_i. We assume an open real-time, or mixed-criticality system in which a number of less critical or best effort threads $\tau_0^B, \ldots, \tau_m^B$ invoke the same components as the real-time threads, thus share the same pools of transient memory. A central goal of TMEM is to observe any tardiness that occurs due to interference from lower-priority threads, and allocate memory to alleviate that tardiness in the future.

The TMEM system of this paper strictly makes no assumptions about knowledge of component worst-case execution time, invocation patterns between components, and other a-priori system characteristics (*e.g.* in [15]). Instead, the system requires the means for measuring each thread's end-to-end tardiness, and information about thread block times in components, and TMEM utilization levels in components (provided via II). However, we do assume that other means are used to prevent the system's overload from real-time thread execution. Specifically, if the tardiness cannot be removed given a hypothetically infinite amount of TMEM, the system design presented here will attempt to minimize, but cannot eliminate tardiness.

A. Interface Between the TMEM Scheduler and Managers

Operation	Description
`unsigned int get_estimated_requests(c)`	Get an estimate of the amount of concurrently requested transient memory
`unsigned int get_thd_blk_time(c,t)`	Get the amount of time a thread spent blocked on contention for tmem in `c`, (b_i^j)
`unsigned int get_thd_blk_cnt(c,t)`	Get the number of times `t` blocked on contention in `c`
`void set_desired(c,amnt)`	Change the desired amount of transient memory, δ^j, for a component

TABLE II
MANAGER OPERATIONS USED BY THE SCHEDULING POLICY.

Each of the transient memory managers provides a number of functions that summarize per-component execution data about threads since the last time the policy was executed. This data is used by the policy to schedule memory. These functions are summarized in Table II.

B. Transient Memory Scheduling Policy

Algorithm 1: Transient memory scheduling policy

```
Input: $T$: Set of real-time threads, $C$: Set of components
while $true$ do
    foreach $c^j \in C$ do
        $r^j$ = get_estimated_requests ($c^j$)
        $ss^j$ = get_thd_ss_requests ($c^j$)
        foreach $\tau_i \in T$ do
            $b_i^j$ = get_thd_blk_time ($c^j, \tau_i$)/
            get_thd_blk_cnt ($c^j, \tau_i$)
        end
    end
    repeat        // While memory can be productively moved
        $c^{min}$ = $\emptyset$
        $c^{max}$ = find_max_tardiness_component ($C$, $T$)
        if $c^{max}$ then
            if $c^{max} \wedge available\_memory > 0$ then
                $available\_memory = available\_memory - 1$
                add_tmem_update_tardiness ($c^{max}$)
            else
                $c^{min}$ = find_min_tardiness_component ($C, T$)
                if $c^{min}$ then
                    remove_tmem_update_tardiness($c^{min}$)
                    add_tmem_update_tardiness ($c^{max}$)
                else
                    $c^{max}$ = $\emptyset$
    until $\neg c^{max}$
    foreach $c^j \in C$ do
        set_desired ($c^j, \delta^j$)
    end
    // Run policy periodically
    periodic_block ()
end
```

We investigate a number of policies, but the most intelligent one attempts to set $\forall_{c^j} \delta^j$ to optimize for $\min(\max_{\forall \tau_i} t_i)$ given each b_i^j.

Algorithm 1 is a greedy algorithm based on this optimization to determine how much transient memory should be assigned to each component, thus fulfilling G3. This algorithm attempts to find the component with the thread that could benefit most (*i.e.* largest decrease in tardiness) from a single unit of transient memory, and the component that would cause the least effect on end-to-end constraints, and moves memory from the second to the first.

To do so, the policy maintains for each component an estimate of what effect adding or removing one unit of transient memory will have on the block time of each thread in that component, thus the effect on their tardiness. We base this estimate for τ_i in c^j on the formula:

$$t\_est_i^j(\Delta) = t_i - \Delta * \begin{cases} b_i^j/(r^j - a^j) & \text{if } r^j - a^j > 0 \\ 0 & \text{otherwise} \end{cases}$$

We use t_i to denote the average tardiness of all jobs that complete since the last time the policy ran. Given a change in memory allocation ($\delta^j - a^j = \Delta$), the policy estimates that the thread's block time will decrease by the an amount proportional to the change in the difference between allocated and requested. Note that when $r^j \leq a^j$, the thread will have no b^j, thus $t\_est = 0$. Note also, that we assume $\Delta \leq r^j - a^j$.

`find_max_component_tardiness` finds the component with the maximum estimated tardiness, $\max_{\forall c^j} \max_{\forall \tau_i} t\_est_i^j(1)$, while `find_min_component_tardiness` finds the component with the minimum, $\min_{\forall c^j} \max_{\forall \tau_i} t\_est_i^j(1)$. Both `remove_tmem_update_tardiness` and `add_tmem_update_tardiness` change a^j and δ^j recompute t_est for each thread that blocked in c^j.

The code shown and described here is simplified to schedule only one type of transient memory. Our general implementation simply treats the number of *virtual components* in the system as $|C| * x$ where is x is the number of different types of transient memory.

The algorithm can move at maximum the number of transient memory units, and the inner loops must find the component, and update tardiness estimates for each thread that blocks in it. Thus the complexity is $O(M|C||T|)$, where M is the number of transient memory units. In practice, we found the overheads in an actual system are small – 0.2% of execution time in our experiments when the policy is run periodically every quarter second. There is a trade-off in determining the policy's periodicity: larger periods mean less overhead due to policy execution, and smaller periods mean the system can more quickly adapt to changes in TMEM requirements throughout the system. We leave this evaluation for future work.

VI. EVALUATION

Unless otherwise noted, all experiments in this section are run on a 3.4 Ghz Intel i7 processor with access to 1GB of RAM. Only one SMT thread (and only one core) is active. We use the hijack [26] technique to boot into COMPOSITE.

A. Microbenchmarks

We report microbenchmarks for our TMEM implementation in Table III. We execute the operations 2000 times, and report the average and standard deviation. Given interrupt interference, it is difficult to obtain a true worst-case – we find worst-case measurements to usually reflect the worst-case interrupt processing path, thus we report the standard deviation of each operation. Invocations include the full call and return path. The thread block and wakeup cost includes a component invocation to the scheduler component to access its interface for thread control. We report results additionally for an Intel Atom 300 series processor running at 1.6 Ghz with only a single core and SMT thread activated.

Operation	i7 Avg	Stddev	Atom Avg	Stddev
Pre-TMEM Invocation	831 (0.24)	2	1291 (0.80)	8
Thread block/wakeup	1740 (0.51)	11	3706 (2.31)	15
Invocation w/ cached stack	860 (0.25)	4	1362 (0.85)	9
Invocation w/ PI	7194 (2.11)	20	13881 (8.67)	49
Cached `cbuf_alloc`	245 (0.07)	6	344 (0.21)	15
Cached `cbuf_free`	317 (0.09)	8	507 (0.31)	31
`cbuf2buf`	304 (0.08)	7	269 (0.16)	10
`cbuf_alloc` w/ PI	7605 (2.23)	37	15180 (9.48)	82

TABLE III
COMPOSITE MEASUREMENTS REPORTED AS "CYCLES (μ-SECONDS)" ON INTEL I7 AND ATOM PROCESSORS.

Discussion. We compare the cost of a normal invocation in COMPOSITE (pre-transient memory) that uses a preallocated array of stacks with the cost using the stack caching mechanism, and find the overhead to be 29 cycles. An invocation that includes a `cbuf_alloc`, `cbuf2buf` in the invoked component, and `cbuf_free` upon return takes 1553 cycles (0.45μ-seconds). `cbuf2buf` might be called in many components as the buffer is passed along. We believe that the current costs are reasonable for zero-copy RPC.

The operations for performing priority inheritance upon transient memory contention are more expensive. These costs are mitigated in many cases using memory scheduling.

B. System Evaluation

In this section, we investigate (i) the effectiveness of TMEM to reduce task tardiness when it is observed while in constrained memory situations, (ii) and TMEM's ability to schedule memory for real-time threads while best effort threads interfere and contend the transient memory. We compare the following system configurations:

`min allocation` – Give each component only the minimal allocation. This demonstrates frequent memory contention, but also the minimal memory configuration.

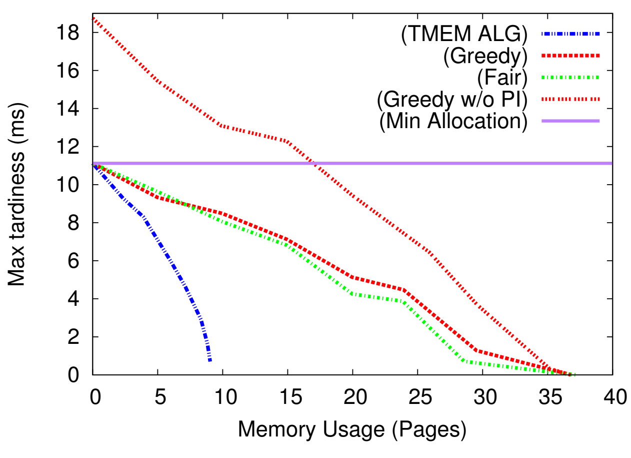
`max allocation` – Execute the system with no memory limits on transient memory and ensure that $\forall c^j, \delta^j \gg a^j$. This avoids contention, but uses maximal memory.

`fair` – Each component is given an equal proportion of the available transient memory.

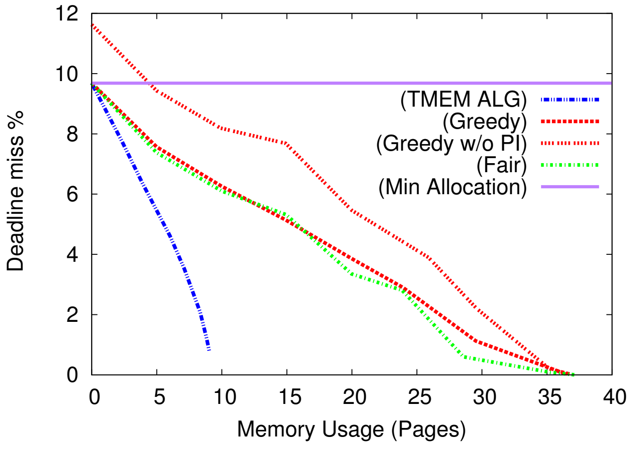
`greedy` – When threads call `tmem_get`, they are always given available memory, until it runs out. We investigate both PI and non-PI variants to compare against a naive implementation.

`TMEM alg` – The memory scheduling algorithm of Section V.

System setup. To tightly control the worst-case execution times of threads (e_i), we specially construct a set of components to accurately measure deadline misses and tardiness for

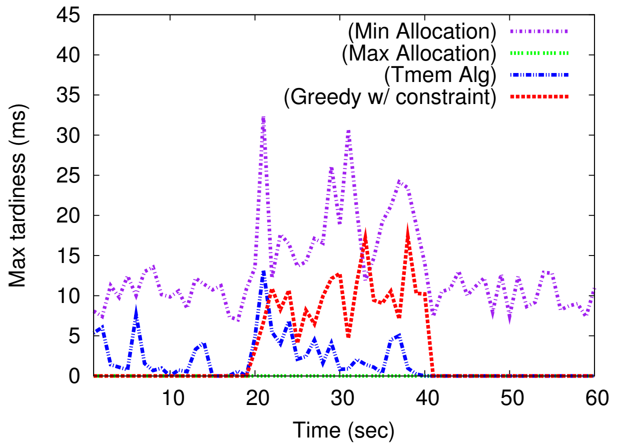


(a)

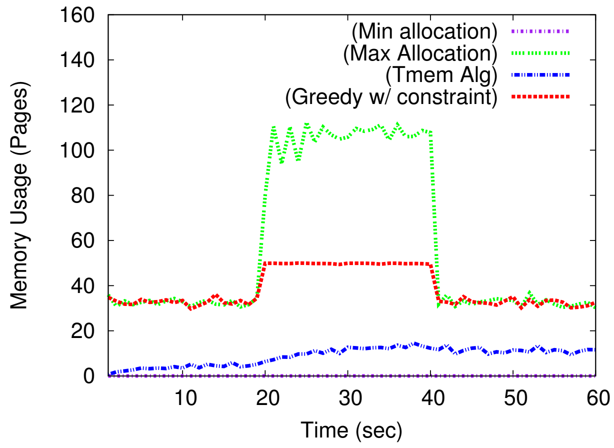


(b)

Fig. 6. Transient memory management effectiveness using different algorithms for memory scheduling. The minimum memory allocation (0 in the graph) to the real-time and shared subsystems is 48 pages.



(a)



(b)

Fig. 7. An open real-time system with best effort threads interfering with and contending for transient memory with real-time threads. Best effort threads execute between 20 and 40 seconds. The minimum memory allocation is 74 pages (0 in the graph).

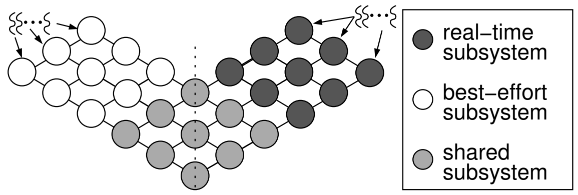


Fig. 8. Experimental setup. The system is configured to consist of three parts: a real-time subsystem with components only invoked by real-time threads, a best-effort subsystem with only best-effort threads, and a set of components shared by both. Components are setup as shown with invocations going downwards. At each component, with a 75% chance, it will invoke toward the dotted line, and 25% away. The three components on the line make invocations with a 50/50 probability. This random factor makes it more difficult to schedule memory compared to a more deterministic system with well-defined heavily used components.

worst-case execution. Figure 8 depicts our experimental setup. Eight real-time threads invoke the two components at the top of the real-time subsystem. When evaluating an open system, eight best effort threads invoke the best effort subsystem.

Each thread makes invocations to "lower" components, and at each, 10% of its e_i is consumed in a tight loop, and the next component is invoked, passing the remaining time left for execution minus the cost of an invocation as an argument to the next component. Threads allocate a pages of CBUFs in each component, and pass one to be mapped into the next component. Once the execution time has been expended, the thread returns to its initial component, and blocks till its next release. The worst-case execution time of a thread is calculated assuming that all TMEM allocations are satisfied from the cache (*i.e.* no contention). Because of this, contention can cause deadline misses and tardiness.

Real-time thread parameters are assigned by first assuming that $\sum_{\forall \tau_i} u_i = 0.8$. p_i is chosen for each thread using an exponential distribution around 100ms. u_i is chosen by also using an exponential distribution around $(\sum_{\forall \tau_i} u_i)/|T|$. e_i is determined as $e_i = p_i * u_i$. The system has a fixed priority preemptive scheduler and uses a rate-monotonic priority assignment. Each experiment is run ten times with different generated task sets, and the averages are reported.

Effectiveness of transient memory scheduling. This experiment executes a system with only real-time threads for 60 seconds. It takes the average of per-second deadlines missed and maximum tardiness of all threads over the run.

Figures 6(a) and (b) plot the maximum tardiness and deadlines missed of the different transient memory management schemes for different maximum amounts of available transient memory. This demonstrates how effectively the algorithms are at meeting real-time targets given limited memory.

Discussion. Memory usage is lower-bounded by 48 due to the minimum allocation required to ensure progress (see Section IV-B). The TMEM algorithm consistently decreases the maximum tardiness of real-time threads for the same maximum amount of memory, compared to the other approaches that don't consider thread end-to-end constraints. A linear regression of the TMEM algorithm and the greedy approach show that the slope of the TMEM algorithm is 4 times steeper. Thus, for each unit of transient memory it is given, it is able to reduce the maximum tardiness by a factor of 4 more than the more naive alternatives. The priority inheritance support provided by the TMEM managers is essential to control the thread's tardiness as is demonstrated by the large gap between the algorithms and their "w/o PI" variants.

Memory scheduling in an open real-time system. Figures 7(a) and (b) plot the different transient memory management methods as the system executes over time. At 20 seconds, the best effort threads are introduced into the system, and at 40 seconds, they complete execution. The best effort threads contend for transient memory in the shared subsystem, complicating the memory scheduling, and making it more difficult to minimize the tardiness of real-time threads. This shows how the memory scheduling policies are able to adjust to changing component concurrency, thus different r^j over time, and attempt to prevent interference of the best-effort threads on the real-time threads.

Discussion. The interference from the best-effort thread significantly impacts not only the memory consumption, but also the maximum latency and deadlines missed of all scheduling algorithms but the TMEM algorithm. When the best-effort threads arrive, they do cause an increase in both the memory consumption, and the tardiness, but the TMEM algorithm is able to both maintain a low memory consumption, and dynamically move memory to components to reduce the tardiness. In contrast, the other algorithms either experience high tardiness or memory usage, and do not as effectively manage the trade-off between the two.

Evaluation of a network RPC service. We evaluate the use of TMEM in a realistic setting. We use a service that answers requests off of a network, processes them, and returns a reply. We configured the system to specifically handle HTTP requests, and return results from a ram-based file system. The system is relatively small, consisting of only 23 specialized components. 13 of these requires stacks from the stack manager, and 7 require CBUFS. Eight threads are used to handle requests on different ports, and one of them, τ_{qos}, is treated as requiring higher quality of service. Here we relax the periodic model used so far, and instead take advantage of the fact that the TMEM policy simply requires thread + component block times, and a thread's tardiness. We interpret any positive variation from the average processing time as the tardiness for τ_{qos}. Table IV shows the resulting response time for τ_{qos}, and the amount of TMEM used in the system. The `static` item denotes a static allocation of all required memory for stacks and shared memory. Such a static allocation is common in many embedded systems.

Policy	Response time (μ-sec)	TMEM above minimum
`min`	38.8	35
`max`	10.1	48
`static`	10.1	140
`tmem` alg	12.7	42

TABLE IV
THREAD RESPONSE-TIME TO GENERATE A REPLY.

Though not detailed here, TMEM deals with task self-suspension, a requirement when threads block waiting for requests off the network.

Discussion. This application shows the generality of the TMEM system. Though requests to such a system are sporadic, the system observes response time variation, and uses memory scheduling and predictable access to TMEM in component caches to control the jitter. A system that scaled larger and required more threads would also require more TMEM.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents the TMEM system for scheduling transient memory to increase effective memory capacity while optimizing for end-to-end application constraints. TMEM employs spatial multiplexing of memory between components to enable efficient access to memory, and predictable temporal multiplexing of memory to reduce the amount of required memory. TMEM focuses on increasing the effective capacity of RAM, while explicitly ensuring predictability in contrast to traditional techniques such as virtual memory. We present an implementation in COMPOSITE with two different forms of transient memory – execution stacks and shared memory buffers – and evaluate the ability of the system to dynamically manage and reduce thread tardiness, and memory requirements. We find that TMEM is able to maintain the efficiency of caches, while lowering both task tardiness and system memory requirements.

In the future we wish to apply the TMEM infrastructure to all memory allocations in the system including those that do not adhere to P2 and the bounded latency between allocation and deallocation. This will enable even `malloc`ed memory to be scheduled along-side execution stacks and shared memory.

Source code for the system can be found on the COMPOSITE webpage at www.seas.gwu.edu/~gparmer/composite.

Acknowledgments. We'd like to thank the anonymous reviewers and the shepherd of this paper. They have significantly improved the quality and presentation of this work.

REFERENCES

[1] J. Liedtke, "On micro-kernel construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.

[2] G. Parmer, “Composite: A component-based operating system for predictable and dependable computing,” Ph.D. dissertation, Boston University, 2009, adviser-Richard West.

[3] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, “The pebble component-based operating system,” in *Proceedings of Usenix Annual Technical Conference*, June 2002, pp. 267–282.

[4] I. Pyarali, M. Spivak, R. Cytron, and D. C. Schmidt, “Evaluating and optimizing thread pool strategies for real-time corba,” in *LCTES*, 2001.

[5] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.

[6] G. Parmer, “The case for thread migration: Predictable ipc in a customizable and reliable os,” in *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT ’10)*, 2010.

[7] W. S. Beebee and M. C. Rinard, “An implementation of scoped memory for real-time java,” in *Proceedings of the First International Workshop on Embedded Software*, ser. EMSOFT ’01, 2001, pp. 289–305.

[8] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, “Redline: first class support for interactivity in commodity operating systems,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI’08, 2008, pp. 73–86.

[9] A. Eswaran and R. Rajkumar, “Energy-aware memory firewalling for qos-sensitive applications,” in *Proceedings. 17th Euromicro Conference on Real-Time Systems, 2005. (ECRTS 2005).*, july 2005.

[10] S. Kato, Y. Ishikawa, and R. R. Rajkumar, “Cpu scheduling and memory management for interactive real-time applications,” *Real-Time Syst.*, vol. 47, September 2011.

[11] P. Druschel and L. L. Peterson, “Fbufs: A high-bandwidth cross-domain transfer facility,” in *Symposium on Operating Systems Principles*, 1993, pp. 189–202.

[12] V. S. Pai, P. Druschel, and W. Zwaenepoel, “Io-lite: a unified i/o buffering and caching system,” in *OSDI ’99: Proceedings of the third symposium on Operating systems design and implementation*, 1999, pp. 15–28.

[13] F. W. Miller, P. Keleher, and S. K. Tripathi, “General data streaming,” in *RTSS ’98: Proceedings of the IEEE Real-Time Systems Symposium*, 1998.

[14] R. Pineiro, K. Ioannidou, S. Brandt, and C. Maltzahn, “Rad-flows: Buffering for predictable communication,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, april 2011, pp. 23 –33.

[15] Q. Wang, J. Song, and G. Parmer, “Stack management for hard real-time computation in a component-based os,” in *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, ser. RTSS 11, Nov 2011.

[16] B. Blackham, Y. Shi, and G. Heiser, “Improving interrupt response time in a verifiable protected microkernel,” in *Proceedings of the 7th ACM european conference on Computer Systems*, ser. EuroSys ’12, 2012.

[17] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, “Timing analysis of a protected operating system kernel,” in *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, Vienna, Austria, Nov 2011.

[18] T. P. Baker, “A stack-based resource allocation policy for realtime processes,” in *RTSS*, 1990.

[19] M.-I. Chen, “Schedulability analysis of resource access control protocols in real-time systems,” Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1991.

[20] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, “Transcendent memory and linux,” in *In Proceedings of the Linux Symposium*, 2009.

[21] G. Parmer and R. West, “Predictable interrupt management and scheduling in the Composite component-based system,” in *RTSS ’08: Proceedings of the 29th IEEE International Real-Time Systems Symposium*. IEEE Computer Society, 2008.

[22] ——, “Hires: A system for predictable hierarchical resource management,” in *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 180–190.

[23] ——, “Mutable protection domains: Adapting system fault isolation for reliability and efficiency,” in *ACM Transactions on Software Engineering (TSE)*, 2012.

[24] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo, “A new kernel approach for modular real-time systems development,” in *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[25] J. Bonwick, “The slab allocator: an object-caching kernel memory allocator,” in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*, 1994.

[26] G. Parmer and R. West, “Hijack: Taking control of cots systems for real-time user-level services,” in *Proceedings of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2007)*, April 2007.