On Spatial Isolation for Mixed Criticality, Embedded Systems

Eric Armbrust, Jiguo Song, Gedare Bloom, Gabriel Parmer
The George Washington University
Washington, DC
{earmbrust,jiguos,gedare,gparmer}@gwu.edu

***Abstract*—This paper addresses some of the challenges of creating a system that enables not only the temporal isolation required for mixed-criticality systems, but also the necessary spatial isolation that enables the decoupling of assurance levels required for different pieces of software. We discuss the application of fine-grained isolation, hierarchical resource management, and the paravirtualization of a legacy RTOSs API, all to enable the system designer to harness memory isolation to control the assurance required for system components.**

I. Introduction

Real-time / embedded system developers face increasing pressure to reduce the size, weight, and power (SWaP) requirements of devices. One solution to reduce SWaP is to package multiple applications onto a single chip and to partition access to the chip resources among the applications. When such applications have differing safety-critical importance, the integration of these applications on a shared platform creates a mixed criticality system (MCS). A problem with MCSs is in sharing resources between applications at different criticality levels, because blocking synchronization primitives can lead to low-criticality tasks interfering with high-criticality tasks. When the MCS is scheduled globally, *i.e.* the same scheduler handles all tasks, solutions based on priority-based synchronization primitives can be applied—for example, criticality-aware versions of the priority inheritance and priority ceiling protocols [1], [2]. However, if the MCS lacks a global scheduler then the job of ensuring that resource synchronization does not cause low-criticality tasks to interfere with high-criticality tasks becomes more difficult. In particular, MCSs that use hierarchical scheduling [3] do not have global scheduling knowledge.

When used with only two levels, a parent scheduler and its children schedulers, a hierarchically-scheduled MCS schedules applications at different criticality levels with distinct children schedulers. The parent scheduler is trusted to schedule the children according to criticality, and each child schedules the application tasks independently. A popular mechanism to support two-level hierarchical scheduling is to use virtualization, with the parent executing in the hypervisor and each child in a guest virtual machine. Applications can be used with minimal modifications and the hypervisor ensures safety of the high-criticality tasks. A problem with hierarchical scheduling with virtualization technology is that performance degradation is prohibitive when children are given small budgets, for example in RT-Xen budgets less than 1 ms lead to untenable scheduler overhead [4]. Another problem with using hierarchical scheduling for MCS is that the existing solutions for task synchronization cannot be adopted easily for resource sharing, because there is no uniform scheduling policy to arbitrate priority and criticality between different schedulers in the hierarchy.

In this paper, we introduce MC-HiRes, Mixed Criticality HIerarchical RESource management, which supports MCSs in the Composite operating system via hierarchical scheduling with the benefits of minimal modification of applications, small performance loss, and resource sharing between applications at different criticality levels. MC-HiRes is a logical extension of our HiRes [3] system with support for MCSs. The primary modification to HiRes is support for a range of mappings of event notification threads (ENTs) to nodes (components) in the hierarchy to provide for strong temporal isolation even in the presence of shared resource synchronization. MC-HiRes avoids the performance problems of virtualization-based hierarchical scheduling by using a guest-aware approach requiring minor modifications *i.e.* paravirtualization. The problem of task synchronization in a hierarchically-scheduled MCS is handled by locating synchronization primitive (locks) in servers that mediate access to shared resources among the clients, which may have different criticalities. Subject to the server having the highest criticality of its clients, MC-HiRes thus solves the two problems identified above for an MCS using hierarchical scheduling.

We demonstrate MC-HiRes by paravirtualizing a legacy RTOS, FreeRTOS, to split RTOS services and application threads into separate components. Isolating application threads enables system designers to assign different criticalities to threads, thus allowing an MCS design without modifying application software. (Slight modification is made to the FreeRTOS kernel.) We evaluate the overhead of using semaphore and message queue services within the FreeRTOS kernel and between application threads and the kernel. Performance loss occurs due to the introduction of spatial isolation along RTOS service calls, but the performance is reasonable with an overhead around 0.25 μ-seconds each time a call is made between criticalities.

Contributions. This paper's contributions include:

- An introduction to the component-based system structuring model, its uses, and the implications for both temporal and spatial isolation of mixed criticality embedded systems.
- A simple extension to our existing hierarchical resource management system (HiRes) to enable hierarchical scheduling for MCS to ensure the mutual temporal/spatial isolation of different criticalities.

*This material is based upon work supported by the National Science Foundation under Grant No. CNS 1149675 and ONR Award No. N00014-14-1-0386. 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 or ONR.

- A system design that provides isolation of mixed-criticality applications from each other for a legacy embedded system that does not provide memory protection, using component-based memory isolation facilities of COMPOSITE.

II. MC-AWARE MEMORY ISOLATION IN COMPOSITE

The COMPOSITE component-based OS provides support for memory protection between separate user-level components based on hardware page-tables. Each component includes local memory for data, code, heap, and stacks that is isolated from other components. Each component exports zero or more interfaces consisting of a number of function call entry-points, and has a set of dependencies on the interfaces exported by other components. Components are passive: only if a thread is explicitly created within the component, or if a thread executing in another component invokes an exported function, will execution occur in a component.

Inter-component communication. Invocations of a function exported by a component result in IPC via *thread migration*. The same schedulable thread executing in the client, resumes execution in the server. After completing its execution in the server, it resumes execution in the client. Control flow integrity is maintained as the server controls its entry-points, thus ensuring that only intended functionality is conducted. Upon entry into the server, sanity checks akin to those done by a traditional kernel in system-call handlers are conducted on parameters. The scheduling context migrates between components (thus scheduling components control thread, not component, execution), it switches between component-local execution state, including execution stacks. These stacks are managed [5] to trade-off the amount of memory they require, and the timing properties of the system if multiple schedulable threads require more stacks than are available in a component (mediated by predictable sharing protocols). In this paper, we assume a simple static allocation of stacks to components commensurate to the number of threads in the system. The sum benefit of this inter-component communication mechanism is that the *end-to-end timing analysis of a thread are identical to those in traditional systems*. This explicitly avoids the dependent-task scheduling problems often encountered by IPC mechanisms that involve coordination between multiple threads. These existing analyses are often pessimistic, especially in a system with large numbers of components.

Resource sharing. Note that resource sharing must still be taken into account, but the prescribed mechanism for this is to mediate all sharing of a specific resource within a critical section within a component. For example, COMPOSITE has a mailbox component that enables multiple threads to communicate via asynchronous message passing. The buffers that hold the data being passed between threads are mediated within the component via a critical section protected by a lock supporting predictable resource sharing. By default we use a lock component that provides priority inheritance. For an MCS, the lock component is problematic because all threads of different criticalities are exposed to each other's interference within the server, and within the critical section. However, *all other execution in other components can be segregated between criticalities*.

Resource sharing mediated by memory-protected components simplifies the analysis of the system. The worst-case hold time of any shared resource (the worst-case critical section length) is provided by the implementation of the mediating component. That component's code is trusted given a combination of the control flow integrity of the IPC mechanism, and its memory isolation. Assuming the component has code appropriate for the criticality of all of its clients, the impact of the resource sharing on each of their timing is *invariant on the clients, and only a property of the mediating component itself*. This simplifies sharing between criticality levels by design, and enables traditional protocols such as Priority Ceiling Protocol to provide predictability, though MC-specific resource sharing protocols can be used (by simply using a different lock component).

Though implementing the sharing of resources in components implies the overhead of communication with that component, round-trip IPC (called "component invocation" here) in COMPOSITE is as efficient as the fastest IPC implementations, and is on the order of 600 cycles on our Intel i7-2760QM CPU running at 2.4Ghz (*i.e.* 0.25 μ-seconds).

A. *Mixed-Criticality-Aware Memory Isolation in* COMPOSITE

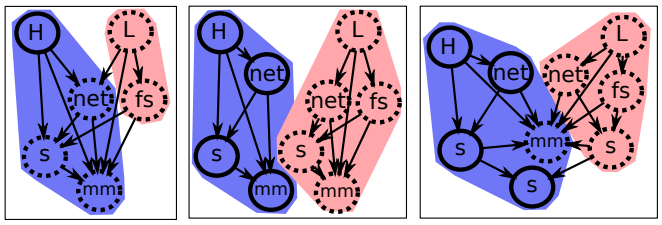

Fig. 1: Example system component structures (S component = scheduler, mm = memory manager). Arrows are component dependencies, and dashed components are harnessed by the low-criticality application. Thus, they require assurance under more complex workloads and must handle more features. The blue region requires a higher level of certification, than the red. Ideally, the blue region would contain only non-dashed, simple components. (a) Traditional structure where high (H) and low (L) criticality applications share all system services. Significant portions of the system requires both full featured support for L *and* high assurance of those components. (b) Separation kernel-style isolation. Only services specialized to H require high levels of assurance, but sharing between H and L is difficult. (c) Selective sharing of functionalities between applications controlling the assurance level required vs. complexity of each component, and hierarchical resource management.

The COMPOSITE thread migration-based inter-component communication mechanism enables the use of traditional scheduling analyses that operate on threads (rather than components, or component-specific threads) and that consider critical sections using predictable resource sharing protocols. The major implication is that thread migration enables the fine-grained decomposition of the system into components, thus strengthening memory isolation. In COMPOSITE, even the lowest-level system services are implemented as components, including: the schedulers, lock managers, time managers, physical memory management, file systems, networking, and drivers. Each component is independently redeployable (assuming its interface dependencies are satisfied), and the entire system can be viewed as a graph of components.

The combination of fine-grained components, the component definition of low-level services, and the use of traditional end-to-end thread timing analysis together provide the ability to explicitly design the structure of the system to mimic that of the criticalities of the different components in the system. The criticality of each component might be static—dependent on off-line tests and analysis of the component's code—or might be dependent on what other component's depend on it, i.e. the workloads for which it is tested. One thing is clear: the high-criticality applications must depend only on high-criticality components. More specifically, the transitive closure over the dependency relation seeded with the high-criticality applications must contain only components that are high-criticality. Importantly, this enables the system to minimize the amount of software that requires high-criticality certification. Whereas in traditional systems that include many system services in the same memory protection domain, all of which must be certified to the highest criticality level, COMPOSITE enables the decoupling of memory and temporal interference between different services.

What is the "correct" amount of sharing between the component graphs for different applications? Figure 1 depicts a simple embedded system with each policy implemented as a separate component. The three different configurations of the system represent (1) a traditional system structure in which applications of different criticalities share many components, requiring that they be certified to the confidence-level of the highest application, (2) a separation kernel-like system in which different criticalities share as few components as possible, and (3) a system with nuanced sharing of components between criticality levels dependent on the sharing relationships and resource availabilities of the system. The benefit of avoiding separation kernel-like share-nothing system organizations is exactly the fact that it is convenient for criticalities to share information and resources between each other (*e.g.* a hard real-time subsystem sharing data to be displayed by an interface). In resource-constrained embedded systems, the extra memory required for separate images and data structures for components replicated between criticalities can be undesirable.

Summary. COMPOSITE enables the straightforward end-to-end analysis of the timing of threads that execute across many components, thus enabling the fine-grained decomposition of the system software into memory-isolated protection domains. Sharing is mediated by service components, and the interference between threads (of different criticalities) is dependent on the mediating component's properties, and is not variant on the contending threads—the critical section interference stays the same, regardless. The criticality and structure of the system's components can be configured to minimize how much software requires high certification, mirror the sharing requirements of applications, and appropriately trade resource usage. This configurability is where the non-traditional, flexible means of constructing a system of components, combined with the fine-grained memory isolation, provides significant benefit for an MCS.

III. MC-AWARE HIERARCHICAL RESOURCE MANAGEMENT

Defining resource management components for a single resource, such as CPU scheduling, across different criticalities is difficult in a system structured as in Figure 1(b). Using CPU management as an example, each scheduler in the system contends for the processor, and the scheduler that should control the CPU at any point in time is not clear. The tension between decentralizing resource management—to customize it for different criticalities, and to increase isolation between them—and deciding at any point which manager to give access to the CPU, motivates our previous work on HIRES [3]. HIRES provides a set of protocols to enable resource manager coordination for hierarchical resource management for CPU, memory, and I/O. In this paper, we focus on extending such protocols to support CPU management and scheduling components for MCSs.

Hierarchical scheduling enables multiple concurrent scheduling components in the system. Though in the simplest case, schedulers form a tree of arbitrary depths, HIRES supports a directed, acyclic graph (DAG) of scheduler structures as well. Each (child) scheduler receives execution time from parent schedulers, and the root scheduler *delegates* all time to its children. HIRES enables each scheduler, regardless of how deep it is in the hierarchy, to dispatch between threads (and even schedule interrupts) with constant and comparable overhead. As each scheduler is implemented as a separate component, they benefit from memory isolation. Thus, possibly complex scheduling policies with dynamic workloads for low-criticality applications can be removed from the certification burden of the high-criticality domain by relying on a simple parent scheduler to mediate between criticalities (e.g., the bottom scheduler in Figure 1(c)).

HIRES Scheduler Coordination Protocols. Parent schedulers delegate to children using a simple mechanism. Parent schedulers are aware of, and uncommonly may dispatch directly, child threads. Normally a parent activates or deactivates a child thread by dispatching to a single *Event Notification Thread* (ENT). That thread executes in a loop delivering event notifications from parent to child. The parent sends a number of notifications: (1) child thread has blocked within an ancestor component, (2) child thread has been activated within an ancestor, (3) a given quantity of time has passed since the last notification—used as a timer for the child scheduler, and (4) the amount of time since the child was last executed (so that it can maintain proper accounting). The child scheduler, after processing all notifications, resumes its normal scheduling behavior and chooses a thread to dispatch.

Child schedulers use the same ENT to send notifications and requests to the parent: (1) thread creation and deletion requests, (2) timeout requests to block until the next notification or given timeout, (3) idle requests (block with an infinite timeout). (For details on the implementation of the protocol for how parent and children synchronize see the original HIRES [3].) The overhead of the ENT and parent-child coordination is on the order of two thread dispatch latencies (to the ENT in the parent, and away from it in the child), plus a single component invocation. The overhead is

around 1700 cycles in total.

HiRes and child thread scheduling. All parent schedulers are responsible for protecting their own data structures, thus they use critical sections. Though parent schedulers rarely dispatch to child threads, in the case of contended resources, parents do not rely on children to mediate the contention. Doing so would put the timing properties of the parent scheduler at the mercy of the child. Thus, in this case of contention, the parent scheduler will switch directly to the child thread that holds the critical section to grant the higher-priority contending thread access. This switch is essential to prevent low-criticality child schedulers from changing the timing properties of the parent scheduler, thus indirectly impacting the timing of high-criticality applications. The policy that the parent scheduler must control its own timing independent of the behavior of any child scheduler is consistent with the resource sharing between criticalities discussion in Section II—the parent makes timing guarantees for critical section length.

Blocking/waking threads in parent schedulers. Given the flexibility of component composition in COMPOSITE, a situation may arise in which a thread managed by a child scheduler will invoke a component that will attempt to block it (*e.g.* due to resource contention). Each component invokes a specific scheduler, and if the service is low-level it might invoke the parent scheduler to block the thread. This situation—and the one originating from the same component waking the thread at a later time—requires coordination between parent and child scheduler. In HIRES, this is where the parent notifications to the child that the thread has been blocked/woken are relevant. Note that this coordination between parent and child scheduler is lacking in user-level threading libraries as monolithic kernels are not aware of user-level threads. Any one thread will block all of them. Systems such as scheduler activations [6] attempt to solve this problem in a manner similar to HIRES. As both parent and child know of the existence of each thread, the HIRES protocols focus on enabling them to coordinate to schedule the threads appropriately.

A. MC-HIRES*: Mixed-Criticality-Aware, Hierarchical Resource Management*

Hierarchical scheduling and mixed-criticality workloads can often be ill-matched [7]. For example, a child scheduler controlling applications of a comparable critical level might need to execute threads at different priorities (there is no fixed relation between criticality levels and priorities). Thus, a single ENT that the parent dispatches to activate the child is insufficient. That single ENT is treated as a single thread by the parent, thus providing abstraction in hierarchical scheduling. This abstraction prevents multiple priorities and criticalities to be attributed to the child.

We generalize the HIRES model into MC-HIRES. MC-HIRES can describe the traditional setup of hierarchical scheduling with all execution in the child abstracted behind the parameters of the ENT, no hierarchical abstraction with one ENT for each child thread, and any configuration of ENTs to children threads in between these extremes. Each child thread is associated with an ENT when it is created, and all notifications for that thread are sent via that ENT. The parent scheduler schedules ENTs as normal threads, thus activating the child according to the parameters of each ENT.

IV. CASE STUDY: LEGACY RTOS MC-AWARE MEMORY ISOLATION

As an example of some of the techniques discussed in this paper, we have modified a popular, simple RTOS to execute in a paravirtualized environment in MC-HIRES, and have used component-based techniques to provide temporal and spatial isolation for MCSs.

A. FREERTOS *Background*

FREERTOS is a simple RTOS used in (deeply) embedded systems for its configurability and small footprint. It is simple and includes basic APIs for thread creation, message queues (synchronous or asynchronous), semaphores, memory allocation, and some facilities for sleeping threads to enable periodic behaviors. Scheduling is fixed priority, preemptive. All threads share the same protection domain, and FREERTOS is meant to execute on the bare metal (though ports exist that execute on POSIX).

FREERTOS is not a RTOS that is MC-friendly. Applications are not spatially isolated from each other, nor is the kernel code and data. Though the system does support temporal isolation between threads with predictable sharing of resources using fixed-priority, preemptive scheduling, it does not provide the necessary memory protection (spatial isolation) to enable the separate certification of code of different criticalities.

B. MC-FREERTOS*: When Virtual is Better than Real*

To enable legacy embedded tasks to execute within the context of an MCS, we provide means for both spatial and temporal isolation for FREERTOS. To accomplish this, we provide a series of modifications to FREERTOS to increase its capabilities within an MC environment, yielding MC-FREERTOS.

First, we paravirtualize FREERTOS to execute within a component in COMPOSITE. This extension is straightforward and involves adding a FREERTOS port that uses COMPOSITE scheduler library functions for switching between threads and for disabling interrupts. The FREERTOS component is a scheduler, thus has permission to dispatch between threads that can migrate between components via invocation. Thread dispatching involves making a COMPOSITE system call. *MC benefits:* FREERTOS and all of its tasks are now spatially-isolated from other components in the system, thus effectively enabling software of different criticalities.

Second, timer interrupts within FREERTOS are implemented using an MC-HIRES ENT. This enables FREERTOS to be integrated into the scheduling hierarchy. Now existing high-criticality tasks can be executed in MC-FREERTOS, while component-based applications with a lower level of assurance can be executed in other subsystems under different schedulers. A root scheduler that implements a simple policy (therefore capable of being high-criticality) schedules the various criticalities. In our prototype, the root scheduler is a simple fixed priority preemptive scheduler with MC-FREERTOS executing at the highest priority. *MC benefits:*

Multiple criticalities can exist in different scheduler subsystems. FREERTOS legacy applications are spatially-isolated (similar to a separation kernel) from other applications, although not from each other.

Third, we paravirtualize the API of FREERTOS to enable multiple criticalities even for legacy FREERTOS applications. Some subset of FREERTOS applications execute in a memory-isolated component, yet still harness the functionality of the FREERTOS kernel. *MC benefits:* Within the MC-FREERTOS environment, multiple criticalities can exist. More importantly, applications can be written that utilize both the FREERTOS API, and can access non-FREERTOS components. The following text describes this technique.

System call API via namespace virtualization. FREERTOS does not have a well-defined system-call layer like OSes that use dual-mode protection. The lack of memory isolation removes the motivation to define such a layer. However, FREERTOS does have a well-defined API, which applications are intended to use, that features functions of the main functionalities (thread manipulation, queue usage, semaphores, timed blocking). The second stage of converting this legacy system into one that is MC-capable is that we paravirtualize this API so that a FREERTOS application can be executed in a separate component (thus separate protection domain). Application code in the FREERTOS application component is identical to that linked into the FREERTOS kernel, except that it is linked with a small FREERTOS-lib that exports the FREERTOS API. That library interfaces with the IPC facilities of COMPOSITE, and invokes functions exported by a FREERTOS-klib (kernel library) linked into the FREERTOS kernel component. The FREERTOS-klib invokes the actual methods within FREERTOS to handle the requests.

The main functions of the two libraries are to (1) marshal arguments between components using the COMPOSITE IPC facilities, and (2) do namespace virtualization. Namespace virtualization does translation between two different namespaces, one in the FREERTOS application, and the other in the FREERTOS kernel. When an application is compiled into the FREERTOS kernel, it shares the namespace with the rest of the system, and most kernel objects (including threads, semaphores) are accessed directly by pointer. However, in MC-FREERTOS, pointers passed from the FREERTOS application *cannot be trusted* to contain a correct pointer. Thus, a set of translation tables exist in the FREERTOS-lib to map from the pointer—expected by the FREERTOS application as part of the FREERTOS API—to an integer descriptor that is passed via component invocation to the FREERTOS kernel. The FREERTOS-klib receives these descriptors, and translates them to the pointers to the corresponding objects within the FREERTOS kernel after validating that the objects are of the correct type for the function being invoked (*i.e.* that the object is used in a well-typed manner). Some details on the main APIs in FREERTOS, and how they are virtualized, follow:

- *Thread management.* The thread creation function must take a callback function to be executed in the new thread. This function pointer is saved in the FREERTOS-lib, and the FREERTOS-klib passes a function that upcalls at a known location into the FREERTOS application, where the thread retrieves the callback, and executes it.
- *Queues.* In addition to the namespace virtualization above, queues must pass data between the FREERTOS application and the FREERTOS kernel. We set up two uni-directional ring buffers of shared memory between the two libraries to pass the data from the FREERTOS application to the kernel, and vice-versa. Queues can behave synchronously, or asynchronously, as determined by the FREERTOS kernel. As the FREERTOS kernel component schedules its threads, including those in the FREERTOS application, with its own consistent notion of priority, we make no changes to the timing properties of the system aside from the overhead for the libraries and component invocation.
- *Semaphores.* These functions are simple and only conduct the virtualization already discussed.
- *Timed blocking.* The function that enables a thread to block for a span of time only requires marshalling the timeout argument to the FREERTOS kernel.

Summary. We present the design of MC-FREERTOS, which is a paravirtualized extension of FREERTOS. FREERTOS is incapable of mixed-criticality execution due to the inability to provide spatial isolation. This is a familiar story for many low-level RTOSes. We paravirtualize FREERTOS to provide a flexible MCS execution environment by porting FREERTOS to a component in COMPOSITE, implementing it in a hierarchical scheduler, and isolating in a separate component the low-criticality threads from those that are high-criticality, thus providing unchanged timing (minus constant overhead factors) and memory isolation.

C. MC-FREERTOS *Overhead and Performance*

Operation	Average	Stddev
FREERTOS Kernel Threads		
Semaphore w/ activation	0.368	0.014
Semaphore, no-contention	0.11	0.000
Enqueue	0.102	0.003
Dequeue	0.103	0.001
Queue round-trip	0.774	0.025
FREERTOS Application Threads		
Semaphore w/ activation	0.708	0.011
Semaphore, no-contention	0.669	0.002
Enqueue	0.418	0.008
Dequeue	0.567	0.009
Queue round-trip	1.699	0.066

TABLE I: Performance of the main MC-FREERTOS functions, both for FREERTOS kernel threads, and for FREERTOS application (low-criticality) threads. All measurements are in μ-seconds.

We measure the overheads of the FREERTOS kernel-resident threads (*i.e.* high criticality threads executing within the FREERTOS kernel component), and the overheads of FREERTOS application-resident threads that must make invocations to the FREERTOS kernel component for service. We execute a number of operations on a Intel(R) Core(TM) i7-2760QM CPU clocked at 2.4GHz. Table I displays these results. They include semaphore operations that activate a waiting thread, thus include the cost of a context switch; the overhead of a pair of uncontended semaphore operations (take + release); the separate cost of asynchronous enqueue and dequeue operations; and the cost of a "ping pong" through

queues which is synchronous round-trip communication between threads. There is no native x86 port of FREERTOS, so we could not compare against that.

Discussion: Adding memory isolation and the incumbent communication overheads between the FREERTOS application and FREERTOS kernel components does have an impact on performance. However, the performance of all relevant FREERTOS functions remains within reasonable bounds. Most overhead is directly attributable to component invocation costs of around 0.25 μ-seconds.

V. RELATED WORK

The most closely related work to this paper is HIRES by Parmer and West [3]. HIRES uses a similar resource hierarchy approach as this paper, which permits delegating memory and I/O in addition to CPU (scheduling), but HIRES does not provide the same strong temporal isolation guarantees in the presence of resource sharing as our work. Thus, HIRES is not directly usable to support an MCS, since the isolation of different criticality levels must be guaranteed.

The prevailing approach to resource sharing for hierarchical schedulers allocates budgets to each child, and then avoids budget exhaustion during critical section execution. SIRAP by Behnam et al., [8] checks for sufficient budget before entering a critical section. HSRP by Davis and Burns [9] permits bounded budget overruns so that critical sections may terminate despite budget exhaustion. Although HSRP can bound the overrun, preempting a critical section in case an overrun occurs is still problematic [10]. A quantitative evaluation of resource sharing approaches is given by Åsberg et al. [11]. Inam et al. modified FreeRTOS [12] to support two-level hierarchical scheduling for an MCS using HSRP to share resources. The authors evaluated the overhead of mode changes but do not examine resource sharing.

As mentioned in section I, hierarchical scheduling with virtualization can support MCSs. Unfortunately, virtualization suffers performance degradation due in part to a fundamental mismatch between mechanisms: virtualization technology was not designed with embedded systems/real-time scheduling in mind [13]. (Bruns et al. [14] argue contrarily that virtualization is effective for MCS on deeply-embedded devices that lack cache and memory management unit hardware. We do not consider such devices.) Prior work in MCSs with hierarchical scheduling attempts to remove the performance degradation while still isolating children [7], [15], [16]. In general, an MCS exacerbates the difficulties of resource sharing with hierarchical scheduling because of the need for strict isolation between different criticalities. The prevailing solution in the literature is to disallow resource sharing. Our work exhibits the strong isolation of virtualization-like approaches for MCSs with low performance loss despite allowing resource sharing.

VI. CONCLUSIONS

The flexibility and configurability of COMPOSITE makes it an ideal platform for MCSs not only because of its temporal guarantees, but also because of the capability to tailor system memory isolation to application criticalities. This paper has examined the use of the component-based model and the associated memory isolation within MC systems to enable configurable spatial isolation between different criticalities. We also approach the problem of extending HIRES into MC-HIRES as a generalization to enable a more descriptive and configurable interface between parent and child schedulers. Finally, we have introduced MC-FREERTOS, which utilizes the memory isolation in COMPOSITE and the hierarchical scheduling from MC-HIRES to enable both temporal and spatial isolation between different criticalities in a legacy RTOS that lacks memory isolation.

REFERENCES

[1] K. Lakshmanan, D. d. Niz, and R. R. Rajkumar, "Mixed-criticality task synchronization in zero-slack scheduling," in *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '11, 2011, pp. 47–56.

[2] A. Burns, "The application of the original priority ceiling protocol to mixed criticality systems," in *1st workshop on Real-Time Mixed Criticality Systems (ReTiMiCS)*, 2013, pp. 7–11.

[3] G. Parmer and R. West, "HiRes: A system for predictable hierarchical resource management," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011.

[4] S. Xi, J. Wilson, C. Lu, and C. Gill, "Rt-xen: Towards real-time hypervisor scheduling in xen," in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT '11, 2011, pp. 39–48.

[5] Q. Wang, J. Song, and G. Parmer, "Stack management for hard real-time computation in a component-based OS," in *RTSS*, 2011.

[6] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism," in *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1991, pp. 95–109.

[7] A. Lackorzyński, A. Warg, M. Völp, and H. Härtig, "Flattening hierarchical scheduling," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '12, 2012, pp. 93–102.

[8] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Sirap: a synchronization protocol for hierarchical resource sharing in real-time open systems," in *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*. New York, NY, USA: ACM, 2007, pp. 279–288.

[9] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, Washington, DC, USA, 2006, pp. 257–270.

[10] M. Behnam, T. Nolte, M. Sjodin, and I. Shin, "Overrun methods and resource holding times for hierarchical scheduling of semi-independent real-time systems," *Industrial Informatics, IEEE Transactions on*, vol. 6, no. 1, pp. 93–104, Feb 2010.

[11] M. Åsberg, M. Behnam, and T. Nolte, "An experimental evaluation of synchronization protocol mechanisms in the domain of hierarchical fixed-priority scheduling," in *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, ser. RTNS '13, 2013, pp. 77–85.

[12] R. Inam, M. Sjodin, and R. Bril, "Mode-change mechanisms support for hierarchical freertos implementation," in *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, Sept 2013, pp. 1–10.

[13] G. Heiser, "The role of virtualization in embedded systems," in *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, ser. IIES '08, 2008, pp. 11–16.

[14] F. Bruns, D. Kuschnerus, and A. Bilgic, "Virtualization for safety-critical, deeply-embedded devices," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13, 2013, pp. 1485–1492.

[15] M. Völp, A. Lackorzynski, and H. Härtig, "On the expressiveness of fixed-priority scheduling contexts for mixed-criticality scheduling," in *1st International Workshop on Mixed Criticality Systems (WMC)*, 2013, pp. 13–18.

[16] Y. Li, R. West, and E. Missimer, "A virtualized separation kernel for mixed criticality systems," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14, 2014, pp. 201–212.