

# FJOS: Practical, Predictable, and Efficient System Support for Fork/Join Parallelism\*

Qi Wang, Gabriel Parmer  
The George Washington University  
Washington, DC  
{interwq,gparmer}@gwu.edu

**Abstract**—With the increasing use of multi- and many-core processors in real-time and embedded systems, software’s ability to utilize those cores to increase system capability and functionality is important. Of particular interest is intra-task parallelism whereby a single task is able to harness the computational power of multiple cores to do processing of a complexity that is untenable on a single core.

This paper introduces the design and implementation of FJOS, a system supporting predictable and efficient fork/join, intra-task parallelism. FJOS is implemented using abstractions that are close to the hardware, and decouples parallelism management, from thread coordination, yielding efficient fast-path operations. Compared to a traditional fork/join implementation, results show that FJOS has less overhead, is more scalable up to 40 cores, and can generally make better use of parallelism. We modify a response-time analysis to integrate system overheads to assess schedulability in a hard real-time environment, and design an effective algorithm for assigning task computation to cores. This assignment more than triples effective system utilization, and when implementation overheads are considered, FJOS maintains high system utilizations, thus providing a strong foundation for predictable, real-time intra-task parallelism.

## I. INTRODUCTION

The increasing prevalence of multi-core and many-core architectures, together with the the stagnation of single-core performance has motivated the investigation of parallelism in real-time and embedded systems. Harnessing inter-task parallelism – where many tasks possibly previously executed on separate systems, are consolidated onto a single multi-core system – has significant research devoted to it. More recently, research has investigated intra-task parallelism – where tasks, themselves are written to harness multiple cores – on real-time systems. This approach increases the functionality and potential of real-time tasks, as they can achieve higher performance than is available on a single core. This can enable tighter control loops, or more intelligence within a given time budget. Intra-task parallelism is particularly relevant in domains with high computation requirements including video processing, computer vision (for which we give an example in Section II), radar tracking, and robotic planning and autonomous vehicles [1].

However, it is notoriously difficult to achieve significant parallelism within a task, especially if using only the lowest-level thread creation and control APIs (*i.e.* provided by pthreads). Frameworks have emerged that provide simpler interfaces to structured parallelism including OPENMP [2] for fork/join parallelism. With OPENMP, different iterations

\*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.

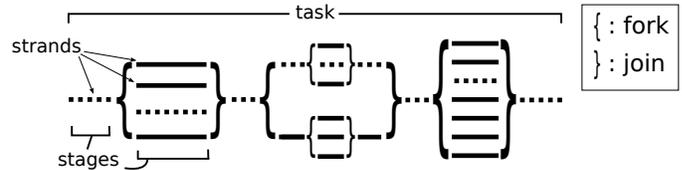


Fig. 1: A fork/join task composed of separate strands of computation, and transitioning through parallel stages, one of which has nested parallelism. The dotted line is a single (master) thread that multiple strands are mapped to.

of a for loop can simply be executed in parallel, where possible. The OPENMP run-time manages the parallelism transparently. Figure 1 shows a single task decomposed into different parallel *stages* (*e.g.* for each loop). Each horizontal line, including each iteration through a loop, represents computation we call *strands*. At a fork, strands are created, and they must all complete execution at a join. Strands are mapped to *threads*, many-to-one, including the strands within the same stage. If there are multiple threads, the task can harness parallelism.

Research into the use of fork/join in real-time systems has focused mainly on the schedulability of such tasks. With the exception of [3], very little research has assessed the practical ability of fork/join infrastructures to support real-time, predictable execution of parallel computation. That work also indicates that significant progress is required, as the overhead imposed for OPENMP is prohibitive for high-frequency tasks. This research focuses on an execution environment we call FJOS, the Fork/Join Operating System, that is designed for predictable, efficient execution of OPENMP workloads. To evaluate the effectiveness of FJOS in hard real-time computation, we take a practical approach by augmenting a response-time analysis with the system overheads, create an overhead-aware assignment algorithm, and demonstrate that we can achieve a high system utilization.

### Contributions and organization.

- **FJOS design and implementation.** Section II provides fork/join background and overview of existing implementations, and in Section III, we present the implementation of FJOS, a system optimized for the predictable and efficient implementation of OPENMP fork/join workloads.
- **FJOS evaluation and comparison.** In Section IV, we compare the overheads of FJOS with Linux’s GOMP, measuring both  $\mu$ -benchmarks, and applications from an established benchmark suite. FJOS demonstrates significant benefit for high-frequency tasks that are common in real-time systems.
- **Overhead-aware response-time analysis (RTA) and assignment.** Section VI introduces an existing RTA that we augment with system overheads from the previous experiments to

determine system schedulability. Additionally, Section VII introduces an assignment algorithm of strands to threads to cores that takes system overheads, especially due to expensive cross socket communication, into account. This assignment algorithm is essential to achieve high utilization.

- *Schedulability evaluation.* Finally, Section VIII presents an evaluation of the effectiveness of both the RTA and the assignment algorithm, before evaluating the schedulability for FJOS and GOMP. This study reinforces the result that FJOS provides significant benefit for both high-frequency tasks, and tasks with significant parallelism.

Sections IX presents the related research, and Section X concludes.

## II. FORK/JOIN PARALLELISM

Fork/join parallelism represents a design point that maps well to the structure of imperative programs: different iterations of `for` loops are executed in separate, parallel threads. When a sequential program reaches a `for` loop, control is forked between parallel threads, and, after the loop, all threads are joined back into a single sequential thread. In OPENMP, the thread that was executing the sequential section before the parallel stage takes part in the parallel execution, and is called the *master* thread. We will call all non-master threads activated at a fork, and that complete strand execution at a join, the *worker* threads. The assignment algorithm in Section VII will discuss the mapping of strands to threads, and will determine the number of threads per stage.

OPENMP is the standardization effort around a fork/join environment. GOMP is a popular implementation of OPENMP on Linux. In specific cases where different iterations of a loop only access the iterator, and local data, C code is trivially converted to use fork/join as depicted in the following code.

```
int i;
#pragma omp parallel for
for (i = 0 ; i < N ; i++) {
    // loop calculations , performed in parallel
}
```

Though this uses the conventional C `for` loop, the `pragma` is interpreted by an OPENMP compiler, which generates closures for different loop iterations, and generates code to invoke a parallel runtime that distributes the computations between threads. In this way, an OPENMP program switches between stages of sequential execution (before a fork), and parallel execution (contents of the `for`); each parallel stage can create more parallelism using nested `omp` loops. The parallelism in a stage is controlled by an environment variable.

Though other abstractions exist for parallelism, we use OPENMP because (1) it is widely-used, (2) it represents a reasonable complexity/parallelism trade-off in that programs can often easily be adapted to use it and harness some parallel execution, and (3) it is mature and well-studied in the real-time and high-performance computing communities.

### A. Intra-task Parallelism Effectiveness in Real-time Systems

To briefly demonstrate the utility of fork/join, intra-task parallelism, we present an application for processing visual

sensor data that could fit into a software stack for autonomous robot control. Images gathered from a visual sensor are passed through a face detection algorithm to detect humans as sources of input and interaction, and as obstacles. We use the `ccv` image processing library (<http://libccv.org/>), and to make use of parallelism, we slice the image into subimages that we pass to the face detector, and merge the results into a final image. We run the system on a 40 core, 4 socket (10 cores per socket) Intel Xeon E7-4850 platform with hyper-threading disabled. If executed sequentially, each execution of the face detector takes 3821ms, but goes down to 331ms and 199ms for 12 cores and 35 cores, respectively.

If a single core is used, the cost of the detectors is far too expensive to include in a reasonable control loop. However, when harnessing intra-task parallelism, the latency for computation becomes tenable for real-time control. It should be noted that this simple form of parallelism decreases in accuracy due to processing partitions of the image. This application leverages the fact that a video stream of images is processed where faces not detected in one frame will be detected in another.

### B. GNU OPENMP: Fork/Join in Linux

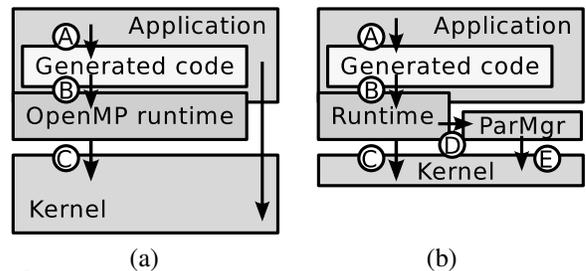


Fig. 2: (a) Architecture of GOMP. (b) Architecture of FJOS. All interfaces in each system are annotated by a circled letter which will be detailed in coming sections.

Here we review the traditional implementation of OPENMP in Linux, GOMP – GNU OPENMP. Figure 2 depicts the architecture of GOMP and FJOS. GOMP consists of a compiler that generates pragma-directed, OPENMP-specific code (A), which invokes the shared run-time GOMP library (B). That, in turn, uses the Linux kernel’s interface (C) to create, coordinate, and synchronize between threads.

**Linux compiler support.** gcc includes OPENMP support (we use gcc version 4.4). When it parses a `pragma omp`, it determines which data is thread-local (for example, the `i` variable in the previous code listing), generates a closure around the work, and synthesizes a block of code corresponding to the loop body. Next, the compiler passes both the generated code and data to the second component of OPENMP, a run-time library in charge of managing parallelism. At the end of the parallel thread’s execution, code is generated to again invoke the library to join the threads. Note that the compiler-generated code is simply linked with separate libraries and run-times for Linux GOMP, and FJOS, and we do not require changes to the compiler.

**GOMP run-time library support.** Table I summarizes some of the most common functions exported by the GOMP

OPENMP run-time lib function	Description
<code>omp_get_thread_num</code>	get the current thread number
<code>omp_get_num_threads</code>	get the number of threads in a parallel stage
<code>GOMP_parallel_start</code>	create parallel threads, execute a function
<code>GOMP_parallel_end</code>	barrier synchronization to join threads

TABLE I: Summary of a subset of the OPENMP and GOMP run-time library functions.

run-time library that correspond to interface (B) in both Figure 2(a) and (b). Each parallel stage has a number of parallel threads identified by their thread number, retrieved with `omp_get_thread_num`. The number of threads in a parallel stage is `omp_get_num_threads`. A parallel for loop, that iterates with a range, will use the thread number and the total number of parallel threads to determine which iterator values to process for a specific thread (i.e. which strands map to which thread). The `GOMP_parallel_*` functions are invoked by the OPENMP-specific, compiler-generated code to fork and join the parallel threads, and the `omp_get_*` functions are called to distribute work (strands) amongst parallel threads.

Though examples in this paper use `omp parallel for`, the functions in Table I are used by many OPENMP operations and are the fundamental building blocks for creating parallelism and joining. FJOS supports all applications in OMPSER [4], unchanged.

**Spin vs. Block in OPENMP.** GOMP includes implementations of thread coordination that utilize spinning and blocking at forks and joins. Parallel worker threads can either spin waiting for an activation via `GOMP_parallel_start`, and the master can spin waiting for all workers to finish, or they can block waiting for corresponding activations. The blocking implementation is Linux-specific, and uses `futexes` (interface (C) in Figure 2(a)): threads blocking waiting for activation use `FUTEX_WAIT`, and are woken when appropriate using `FUTEX_WAKE`. The implementation of this system call will use inter-processor “rescheduling” interrupts to awaken threads across cores. GOMP uses spinning unless either an environment variable is used to request blocking, or if the number of threads is greater than the number of cores. Though the spin-based implementation is faster, it requires parallel threads to consume processing time even when not doing useful computation, thus possibly wasting power, and preventing other task’s execution. It is ideally suited for environments such as high-performance computing where a system is dedicated to a single task, and power is (until recently) a secondary concern.

**OPENMP scheduling.** OPENMP supports multiple means for scheduling work between parallel threads. If a parallel loop includes more iterations (strands) than there are threads, this scheduling mechanism is important. OPENMP supports *dynamic* and *static* scheduling policies. Whereas dynamic policies will attempt to choose a thread for each separate work item, static policies partition the iterations at fork-time (using the `omp_get_*` functions). Static scheduling is the default. Dynamic scheduling is useful when the variability in the processing time for each work item (strand) is large, thus enabling the fine-grained balancing of work. This often imposes overhead as inter-core coordination becomes frequent

(especially in the worst-case). However, in real-time systems, each iteration of the same code has a comparable worst-case execution time, thus increasing the utility of static scheduling. We, therefore, focus on static scheduling in this paper. This focus is consistent with much [1], [3], [5], [6], [7], but not all [8], fork/join real-time research.

Though OPENMP generally is an implementation of fork/join parallelism, it also provides the `nowait` construct which avoids the join at the end of the parallel construct. Though we support this construct, for the sake of space, we do not discuss it further.

**OPENMP for real-time systems.** OPENMP does provide some useful support for real-time systems including controlling the maximum number of threads, setting thread affinity, and disabling thread migration. However, OPENMP has a few impediments to its adoption in real-time systems. (1) It does not provide fine-grained control over the priority of individual threads. (2) It does not provide access to synchronization primitives (i.e. critical sections) with predictable resource sharing protocols [9]. We consider these additions to the OPENMP runtime to be relatively straightforward (i.e. widening the OPENMP API). FJOS does not have these shortcomings.

### III. FJOS: SYSTEM SUPPORT FOR PREDICTABLE, EFFICIENT FORK/JOIN

FJOS is an operating system built to support predictable OPENMP computation. Unlike GOMP, FJOS is a clean-slate implementation that focuses on predictability, simplicity, and, lastly, performance. FJOS achieves these goals by (1) using only *wait-free synchronization between coordinating threads*, thus avoiding any locking, (2) using a *low-level, optimized API for asynchronous, cross-core notifications* with multi-cast Inter-Processor Interrupts (IPIs), thus enabling efficient blocking fork/join, and (3) *decoupling the thread scheduling, prioritization, and core assignment, from the common-case fork/join operations*. Much of this is possible as FJOS is implemented as an OS on the COMPOSITE component-based system.

**COMPOSITE background.** COMPOSITE is a component-based OS in which system policies and most abstractions are defined in fine-grained, user-level components. Components in COMPOSITE consist of code and data that implement some functionality and export a functional interface through which other components can harness that functionality. Components have a set of functional dependencies on interfaces that must be satisfied by other components. Components execute at user-level in separate hardware-provided protection domains, and access to resources and communication channels is restricted by a capability system. Even low-level services such as scheduling [10], physical memory management and mapping, synchronization, and I/O management are implemented as user-level components. Invoking a function in the interface of a depended-on component transparently triggers thread-migration-based [11] synchronous inter-component communication (called “component invocation” here). In this way, the same schedulable thread executes through many components,

and can be preempted at any time. Multiple threads can concurrently execute within a component and predictable resource sharing protocols are required [9] just as they are in system services of more traditional OSes. Components are passive unless a thread is created in them, or a thread from another component invokes the component.

**FJOS overview.** FJOS is implemented as a set of components and libraries in COMPOSITE. See a depiction of the software in FJOS in Figure 2(b). It depends on a small set of other components (namely, a scheduler and physical memory mapper), thus enabling a lean system with a (relatively) simple software stack. That said, the system is not customized to the point of only running OPENMP tasks. General tasks spanning from best-effort and general purpose to hard real-time with specific time management policies, are supported. The mutual customization of different execution environments is supported by HIRRES [12]. In FJOS, we reuse a scheduler and memory manager from the existing COMPOSITE code-base. Currently, we have only investigated using fixed-priority, preemptive scheduling. However, as schedulers are replaceable user-level components in COMPOSITE, another scheduler could be plugged in.

COMPOSITE supports parallel execution using *partitioned scheduling* whereby a thread created on a core cannot migrate later to another core. This simplifies the implementation of the scheduler as it avoids cross-core synchronization on scheduler data-structures, and the overheads of migrating thread working sets between caches. Threads executing on different cores can invoke the same components (similar to how multiple parallel Linux threads can invoke the kernel), thus data within a component must be either partitioned (as with the scheduler), protected via locks that use predictable resource sharing protocols, or use the appropriate non-blocking synchronization.

<code>int acap_trigger(cid_t)</code>	trigger an event
<code>int acap_wait(cid_t)</code>	block current thread until the event is triggered
<code>cid_t acap_receiver_crt(comp_t)</code>	create a wait-able cap
<code>cid_t acap_sender_crt(cid_t, comp_t)</code>	create a trigger-able cap; associated with a receiver cap

TABLE II: ACAPs: kernel asynchronous communication API.

**Asynchronous notification in COMPOSITE.** In addition to synchronous component invocations which mimic function calls, FJOS adds general support for asynchronous event notification between threads. The kernel-provided API for this functionality is summarized in Table II. The functions operate on *capabilities* that reference asynchronous end-points (we call these ACAPs). The first two functions are invoked to both block a thread waiting for an event (`acap_wait`), and trigger that event which will activate a blocked thread (`acap_trigger`). No data is passed between the trigger and the wait, only the event notification. These two functions comprise interface (C) in Figure 2(b).

The thread that triggers the event can be on a separate core than the thread that is woken up, effectively enabling events to actively be sent between multiple cores. This cross-core thread activation is implemented using the processor’s Inter-

Processor Interrupts (IPIs). ACAPs provide efficient wakeup of blocked threads across cores, enabling threads to block waiting for events (*i.e.* forks). Thus cores execute other tasks in the mean time, and can be put into low-power states to save energy. The downside of blocking is that IPIs and scheduling overheads are higher than only using shared memory with spinning. A received IPI that activates a thread waiting on an ACAP results in a scheduling decision. The execution time of an IPI is bounded, and ACAPs can be used to limit the rate at which IPIs are generated within a hard real-time system. Though the IPI itself executes at the highest priority, the resulting computation is properly prioritized using COMPOSITE’s support for user-level, component-based scheduling. See the Appendix for more information. An alternative design would unify the interrupts and task priority namespaces [13]. However, to maintain implementation simplicity and generality, we instead focus on minimizing, and properly accounting for the IPI execution.

**The PARMGR: parallelism management in FJOS.** The second pair of functions in Table II are used by a separate component, the parallelism manager (or PARMGR) and constitute interface (E) in Figure 2(b). This component has the access rights to create the ACAP endpoints. Specifically, it creates the ACAP that is receiving the event notification with `acap_receiver_crt`. After this function returns, the current thread (making the system call) can now wait on the ACAP in the specified component (the `comp_t` argument). The common-case use of this is for a thread wishing to coordinate with another thread via ACAPs, to invoke the PARMGR to create the end-point. One or more sender ACAPs can be created with `acap_sender_crt`, that are connected to the same receiver ACAP – the first argument.

The PARMGR is, more broadly, in charge of managing the assignment of computation to cores at the proper priority, and appropriately mapping the ACAP-driven coordination between cores. It does this for all components and threads in the system. It takes an assignment of strands to threads, and threads to priorities and specific cores, from the assignment algorithm in Section VII, and deploys threads and ACAP linkages between threads accordingly. The PARMGR creates threads appropriately by harnessing the scheduler component’s thread creation mechanisms (that, in turn, use thread creation system calls). In FJOS, the assignment and structure of the intra-task coordination is orchestrated by the PARMGR, leaving the OPENMP run-time library to be concerned only with fast-path operations for fork/join. This separation of concerns, enables the PARMGR to be replaced, thus completely modifying the structure of parallelism, if desired.

Currently, FJOS does not support any of the `omp_set_*` functions that are often used to control, for example, how many parallel threads are used for an application. Instead, the PARMGR manages parallelism across all tasks to ensure *full-system* predictable execution. If desirable, these `omp_set_*` functions could be supported by widening the PARMGR interface.

**FJOS run-time library.** Each thread involved in fork/join

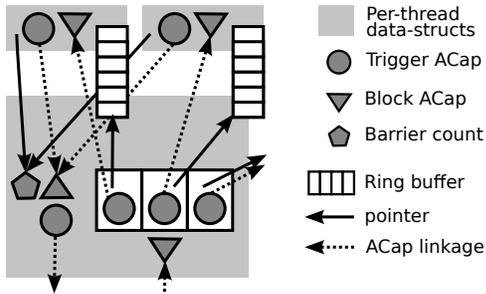


Fig. 3: FJOS run-time library data-structures.

computation in FJOS has a set of per-thread data-structures used for inter-thread coordination. Figure 3 depicts those data-structures. Each thread has: (1) an array of triggerable ACAPs and wait-free ring-buffers<sup>1</sup>, one per worker thread, that the master will use when activating and passing data to these workers upon fork; (2) workers include an ACAP to block on awaiting a trigger from the master to fork, and a pointer to the ring-buffer to contain the function/data to process; (3) a pointer to a barrier count variable that is used to join to track how many workers have joined, and an ACAP to trigger if we are the last worker; and (4) an ACAP to block on if we are the master, and awaiting the completion of a join. The only data-structures that are shared between threads (thus cores) are wait-free – the ring buffer and the barrier count, thus we do not require any locks. The ring-buffer is used for a master to transfer to a worker both the function pointer to the compiler-generated code for the computation, and data<sup>2</sup>. The barrier count variable is initialized before a fork to the number of parallel threads, and each thread atomically decrements it when they join. The last to do so, sends a notification to the master to resume execution. The data-structures are modified with simple fetch-and-add atomic instructions. As all shared memory operations involved in inter-thread coordination are wait-free (*i.e.* they have local progress guarantees), they have a fixed (and low) worst-case cost on a given system, thus support hard real-time execution.

The FJOS thread’s data-structures are composable, and naturally support nested fork/join, inter-component coordination (given the ring buffer and barrier counter are shared between components), and parallelism beyond fork/join. Such support requires changes only to the PARMGR.

Note that FJOS does not modify the compiler, and instead provides only a customized run-time environment. The FJOS OPENMP library exports functions including those listed in Table I that are linked with compiler-generated application code.

**PARMGR interface.** Table III summarizes the interface the PARMGR exports to components that use OPENMP in FJOS, (D) in Figure 2(b). These functions are harnessed at application initialization, or could also be used on-demand. They enable the creation of the per-thread structure in Figure 3. `parmgr_fork_num` is invoked by all new fork/join threads, and returns the number of worker threads that execute in

<code>int parmgr_fork_num()</code>	# threads this master forks
<code>fork_info_t parmgr_fork_info(int)</code>	retrieve information about a specific worker thread to be forked
<code>join_info_t parmgr_join_info()</code>	information a master thread uses to join
<code>fork_info_t parmgr_worker_info()</code>	information a worker thread uses to await activation, and execute
<code>join_info_t parmgr_worker_barrier()</code>	information a worker thread uses to join

TABLE III: PARMGR functions to coordinate parallelism. `fork_info_t` and `join_info_t` are defined in the text.

fork/join stages with it. For a worker that is not a master, this number is zero. `parmgr_fork_info` returns the triggerable ACAP and ring-buffer information (in `fork_info_t`) about this worker that the master uses to populate its worker array. `parmgr_join_info` provides a master with data needed to join: the ACAP to block on (to be triggered by the *last* joining worker, and the barrier count variable). Each worker thread uses the second part of the API to retrieve the ACAP to block awaiting a fork (with `parmgr_worker_info`), and to retrieve the ACAP to trigger, and barrier counter for the master.

**Fork/join spinning.** The above description is simplified to describe fork/join using a blocking mechanism. This is an important policy as it enables other tasks to execute in the mean time. However, FJOS, also supports inter-thread coordination using spinning. The PARMGR controls this by simply returning a specific FJOS\_SPIN value as the ACAP. In this case, all worker threads simply spin awaiting data, thus activation, in the ring-buffer. Comparably, when joining, the master simply spins when it joins on the barrier counter integer that counts worker completions till it is zero, then it continues. Though spinning will, in general, have better performance characteristics than blocking, blocking is desired when (1) multiple real-time tasks share a core, (2) energy consumption is of concern, (3) best-effort tasks wish to use spare capacity.

**Fork multi-cast optimization.** The master threads in the initial implementation of FJOS made an array of  $Q$  = number of cores,  $\langle$ ACAP, ring-buffer $\rangle$  pairs, and successively triggered each ACAP, where  $Q$  was the parallelism of the stage. Though the  $O(Q)$  cost of this operation is acceptable on a single socket, as we went to multiple sockets, each `acap_trigger` caused cross-socket IPIs that are particularly expensive. Thus the current implementation of FJOS instead uses a multi-cast structure to delivery the IPIs. One *dissemination* thread per task, per socket (except on the master’s socket) is activated by an ACAP from the master. The dissemination thread then activates the ACAPs for threads on the cores within its socket, including any worker thread on the same core as the dissemination thread. These IPIs are now socket-local, and have a much lower latency. This optimization was carried out almost entirely within the PARMGR by adding the dissemination threads as intermediate “workers” activated by the master, that activate the real workers. This is similar to the methods used in [14] for TLB shutdown, but in that case, they use shared memory and spinning.

The dissemination threads execute at the highest thread

<sup>1</sup>Modified from the Concurrency Toolkit, <http://www.concurrencykit.org>.

<sup>2</sup>Unless the OPENMP `nowait` construct is used, there will only ever be one item in each ring buffer.

priority and immediately block after triggering the workers. This is to avoid delaying the activation of the workers on the other cores on the dissemination thread’s socket. If this were not done, then any delays in the execution of the dissemination thread would impact the activation time of *all* workers on that socket. The trade-off here is that now these dissemination threads must be taken into account in the timing analysis as high-priority threads. We found that their maximum overhead for a fork is 14.10  $\mu$ -seconds on our system.

Though we tried comparable optimizations for the spin-based implementations of fork and join, they did not provide better performance in the average or worst case than the naive implementations. This might change if the number of sockets increased greatly, but we have not tested past 4 sockets with up to 40 cores.

**Summary.** FJOS has multiple interfaces and layers that interact to implement the OPENMP specification. The emphasis of this implementation is on decoupling the management of priority, assignment, strand to thread partitioning, and the structure of thread interactions, from the common-case operations involved in fork/join. This separation of concerns has enabled the configuration of parallelism management (by adding dissemination threads), and the efficient and predictable thread interaction – mediated by wait-free data-structures, and a low-level, secure wrapper around IPIs. Notably, the entire system including kernel and essential components is less than 15K lines of code (the PARMGR and run-time library are less than 1500). This might open up the use of parallelism for systems that have stringent certification requirements.

#### IV. FORK/JOIN SYSTEM EVALUATION

We evaluate FJOS by comparing it to GOMP on both vanilla Linux and Linux with the real-time patches applied (we’ll refer to this as LinuxRT). Some unexpected results on LinuxRT motivate the inclusion of vanilla Linux. Both Linux systems use kernel version 3.10.10. We compile all code with gcc version 4.4. All experiments are conducted on a system consisting of Intel Xeon E7-4850, 10 core chips, with four sockets. Hyper-threading is disabled, leading to a total of 40 cores. We boot and execute COMPOSITE using the Hijack technique [15].

**Microbenchmarks.** For the real-time execution of fork/join workloads, we require an understanding of the underlying overheads of the different operations for any OPENMP framework, so that they can be integrated into a schedulability analysis. We execute an empty parallel for loop using five different system configurations: (1) FJOS using spinning, (2) FJOS using blocking, (3) GOMP on LinuxRT using spinning, (4) GOMP on LinuxRT using blocking, and (5) GOMP on Linux using blocking. Note that the implementation and results for GOMP using spinning on Linux are identical on LinuxRT (*i.e.* no system calls are made). Figure 4 shows the results of measuring the code on  $N$  cores:

```
#pragma omp parallel for
for (int i = 0 ; i < N ; i++) ;
```

The time is measured with rdtsc, the time-stamp counter, and each measurement is performed 100,000 times. Figure 4(a) reports the average, while Figure 4(b) shows the maximum. For these measurements, we filter out interference from the timer interrupt, as the worst-case of the timer should be considered separately.

We assign threads to cores in a manner that minimizes the number of sockets involved. As this doesn’t produce a worst case for *all* assignments, we instead change the assignment algorithm to also minimize the number of sockets involved. This has a slightly negative impact on schedulability if no overheads are considered, but enables much more efficient run-time execution, thus increased practical schedulability.

**Discussion.** First, the spinning implementation in FJOS has significantly less overhead than the spinning implementation in GOMP. This is due to the avoidance of any shared data-structures besides the appropriate ring-buffers, thus removing locking overheads (note that GOMP does not use a lock with a predictable resource sharing protocol, but this extension should be straightforward). In fact, the blocking implementation in FJOS is not much slower than the spinning implementation in GOMP especially for higher core counts. However, the blocking implementation in vanilla Linux shows significantly higher overhead. Using shared data-structures, and futexes to block and wakeup-threads imposes more overhead than the minimal structures in FJOS. For example, futexes are a very general mechanism that supports both notification and synchronization, and imposes many overheads including looking up the kernel futex data-structure based on a user-level address. On the other hand, ACAPs in FJOS are specialized, light-weight wrappers that provide access control around IPIs. The outlier in these graphs is the blocking implementation of LinuxRT. The worst-case overhead is almost a millisecond at 40 cores. This result is consistent with the overheads observed in [3], and we were not able to determine the cause. We include the vanilla Linux result for perspective.

**Application studies.** We choose three programs from the OMPSCR benchmarking suite [4] that are relevant to real-time systems. Most benchmarks not included are relevant within high-performance computing or don’t have much relevance for embedded systems, for example, computing Pi or producing the Mandelbrot sequence. We vary the input to each application to assess the impact of system overheads. These three applications are the Fast-Fourier Transformation (FFT) that is pervasive in signal processing, solving a partial differential equation using the Jacobi iterative method, and the LU decomposition.

Figure 5 shows the performance of each of these applications for different input sizes for each of the implementations. The plots represent the average of 100 runs.

**Discussion.** If the computation within a parallel stage is large enough, then the overheads of the OPENMP implementation matter less, and, generally, the scalability of the application is better. That is, as more cores are allocated to the application, it proportionately gets faster. However, as the amount of computation in the parallel stages decreases,

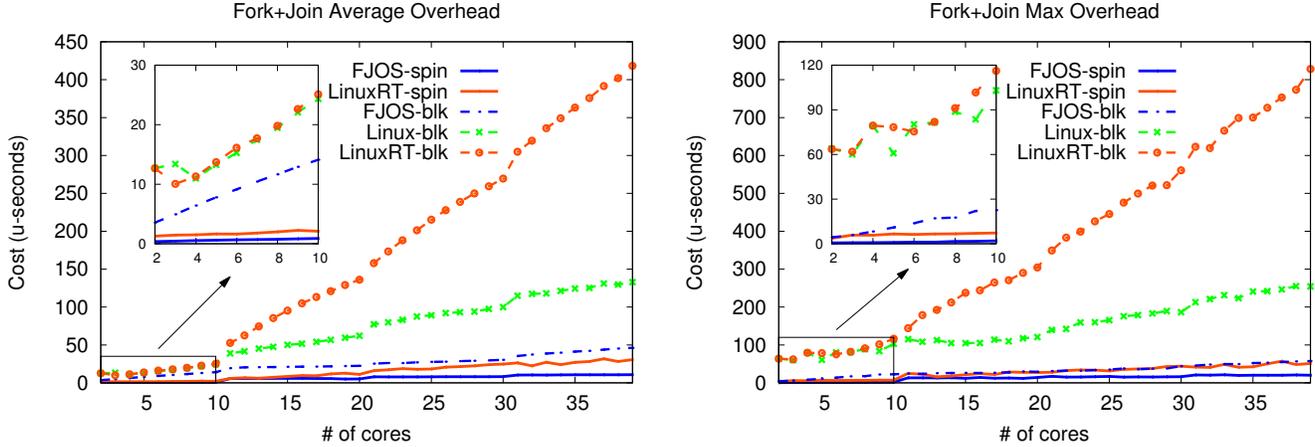


Fig. 4: Microbenchmarks for different implementations of OPENMP. Plotted are average and maximum values where each point represents 100K runs. Note that the Y-axis is different for each graph.

the relative proportion of the OPENMP implementation’s overhead dominates. We note that for real-time systems that might use the computation’s results for control, or for sensor processing, it is not uncommon to execute at a frequency from 30Hz all the way up to 1000Hz, naturally increasing the impact of any overhead. When these applications are executed at high frequencies, FJOS is most useful. Interestingly, the FJOS-block results often suffer only a small degradation, and are sometimes better than LinuxRT-spin. Even though FJOS-block avoids wasting computation, it still provides reasonable parallel performance.

Note that one major irregularity in these graphs is that for larger task sizes for Jacobi and LU reduction, tasks have an increased speedup after decreasing. We believe this is due to the increased availability of cache as more sockets are used. For a worst-case timing analysis, either the worst-case cache allocate has to be assumed, or an advanced cache-analysis needs be performed. This is beyond the scope of this paper.

## V. FORK/JOIN SYSTEM MODEL

So far, we have discussed general support for fork/join in real-time systems. The implementation has applied equally to hard real-time, soft real-time, and even best-effort systems. We wish to provide an analytical basis to determine the schedulability of FJOS task sets, thus we focus the rest of this paper on a specific task model.

A system consists of a set of tasks  $\{\tau_0, \dots, \tau_{n-1}\} \in T$ . We assume a simple periodic task model in which each task consists of an infinite stream of jobs. Each task  $\tau_i$  has periodicity  $p_i$ , with implicit relative deadline, *i.e.*  $d_i = p_i$ . We adopt a simple fork/join execution model that is similar to the one in [7]. For a task  $\tau_i$ ,  $s_i$  is its number of stages.  $\tau_{i,j}$  represents the  $j$ th stage of  $\tau_i$  when  $j \in [0, s_i)$ .  $n_{i,j}$  is the number of parallel strands in  $\tau_{i,j}$ .  $\tau_{i,j}^k$  is the  $k$ th strand of  $\tau_{i,j}$ , if  $k \in [0, n_{i,j})$ . When  $s_i = 1 \wedge n_{i,0} = 1$ ,  $\tau_i$  is a sequential task. For simplicity, we currently don’t consider nested parallelism. In order to analyze task response time, the worst-case execution time (WCET) of each strand is required, and represented as  $c_{i,j}^k$ . The WCET of  $\tau_i$  is  $c_i^{max}$ , which is the execution time required by  $\tau_i$  when only one core is

dedicated to it. The critical path length (CPL) of  $\tau_i$  is  $c_i^{cpl}$ , which is the execution time required when infinite number of cores is assumed and all strands execute on different cores with no system overheads. We focus on hard real-time systems where all task deadlines must be met. Necessarily,  $c_i^{cpl} \leq d_i$  as it’s essential for a system to be possibly schedulable. The utilization of task  $\tau_i$  is  $U_i = c_i^{max}/p_i$ , which can be greater than 1 in which case parallelism must be harnessed. The number of cores on the system is  $Q$ . We assume fixed-priority, preemptive, partitioned scheduling. The set of tasks with higher-priority than  $\tau_i$  and the set with lower-priority are  $hp(i)$  and  $lp(i)$  respectively. All strands of a task share the task’s priority, and there is no inter-strand interference: multiple strands for a stage on the same core execute sequentially in the same thread.

## VI. OVERHEAD-AWARE RESPONSE-TIME ANALYSIS

We adapt the response-time analysis from [7] to our system, and add system overheads to conduct a schedulability analysis for parallel hard real-time systems. In that work, a general event model is used to describe task arrivals. As we assume a simple periodic task activation model, significant portions of the RTA are simplified.

We briefly summarize the RTA introduced in [7]; for additional details, see that paper. The response-time of a task is the maximum amount of time it takes to complete from its activation given interference from other tasks, and it is analyzed stage by stage. For task  $\tau_i$ , the analysis starts from its first stage  $\tau_{i,0}$ . Within a stage, the execution of strands of  $\tau_{i,j}$  that are assigned to different cores is analyzed separately. Without loss of generality, we consider each core to have only one strand of  $\tau_{i,j}$ : when multiple strands of a stage are assigned to the same core, they are combined into an *aggregate strand* that is analyzed as a single strand. For each strand, interference from strands of higher priority tasks is taken into account by finding their maximum overlap with the strands (based on period, execution time and jitter). Once all strands in  $\tau_{i,j}$  are analyzed, their largest response time decides the entire stage’s response time.  $\tau_{i,j}$ ’s stage-completion time, which is represented as  $B_{i,j}$ , is defined as the response time

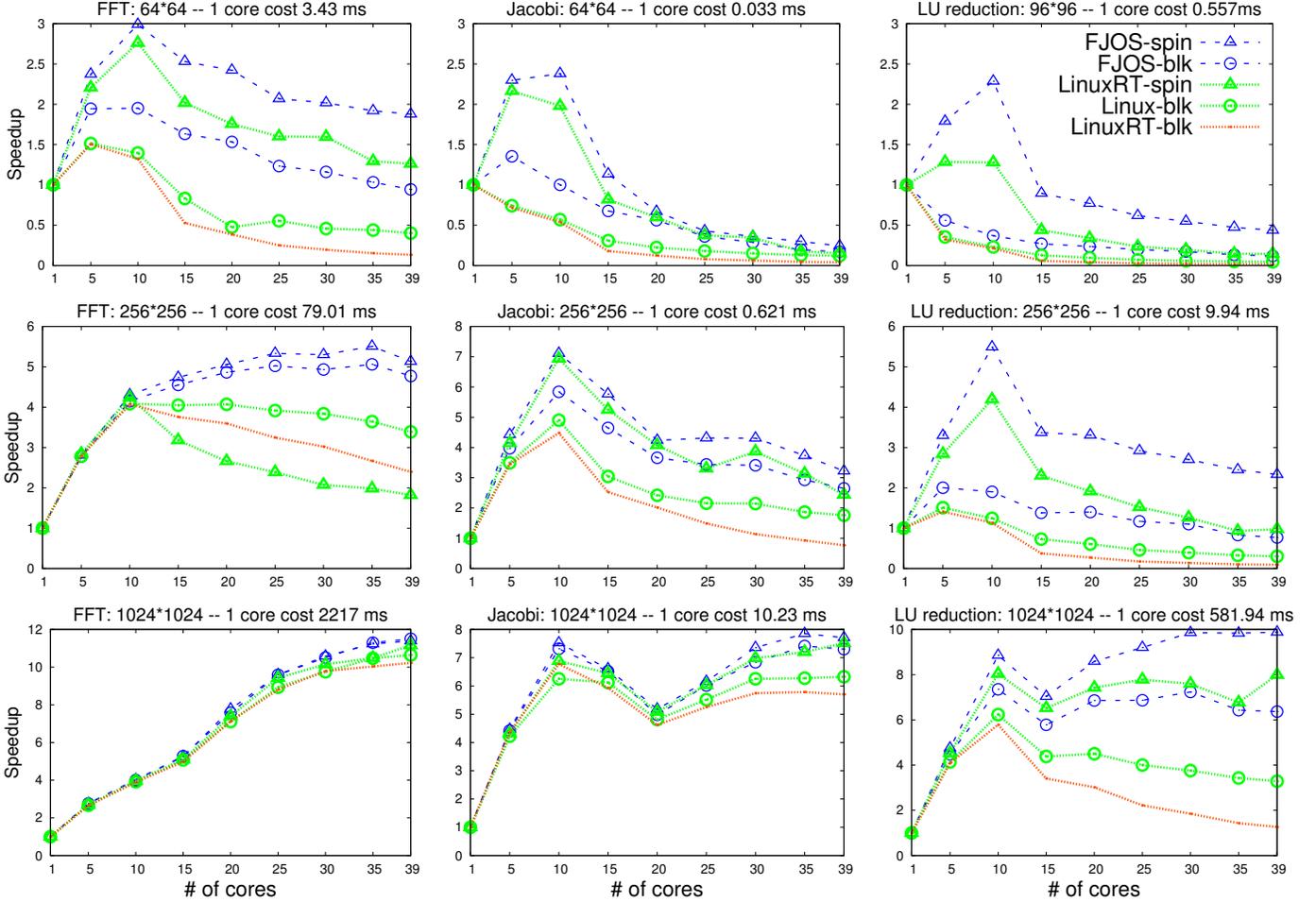


Fig. 5: Three applications from the benchmark suite in [4]. We plot differently sized matrices to show different computation time to fork/join overhead ratios. Applications compute the Fast Fourier Transform, solve equations using the Jacobi iterative method, and do LU decomposition. The speed of sequential execution is reported in the graph titles, and Y axis shows speedup over sequential execution.

of  $\tau_{i,j}$  offset from the activation time of  $\tau_i$ . Therefore, the start time of  $\tau_{i,j+1}$  is  $B_{i,j}$ . The completion time of the last stage is the task response time. More precisely,  $B_{i,j}$  is used as the release jitter of the strands in  $\tau_{i,j+1}$ . The strands in the first stage have 0 jitter. Combining with  $p_i$  and  $c_{i,j}^k$ , the jitter of  $\tau_{i,j}^k$  is used to derive the worst-case interference from  $\tau_{i,j}^k$  to strands belong to  $lp(i)$ .

To reduce the pessimism of the analysis, the RTA in [7] performs an optimization to remove redundant interference from higher-priority tasks across stages. When analyzing a stage  $\tau_{i,j}$ , the strand that contributes to the worst case response time of  $\tau_{i,j}$  is considered to be the *critical strand* of the stage. The core that a critical strand  $\tau_{i,j}^k$  executes on is the *critical core* of the stage. The interference from  $hp(i)$  to  $\tau_{i,j}^k$  on the critical core is recorded. In the analysis of future stages, this recorded interference is prevented from identically impacting those stages. For example, if 1)  $\tau_{i,j}^k$ ,  $\tau_{i,j+1}^l$  and  $\tau_{u,v}^w$  (which belongs to  $hp(i)$ ) are all assigned to the same core, 2)  $\tau_{i,j}^k$  records a specific amount of interference from  $\tau_{u,v}^w$ , and 3) by  $\tau_{i,j+1}^l$ 's iterative response time (relative to  $\tau_i$ 's activation) the amount of interference  $\tau_{u,v}^w$  can generate is no more than the amount  $\tau_{i,j}^k$  records, then  $\tau_{i,j+1}^l$  does not need to consider interference from  $\tau_{u,v}^w$ . Meanwhile, on the

non-critical cores, the interference from  $hp(i)$  is “delayed” to future stages, where it can contribute to critical path in these stages. This guarantees the worst-case scenario is considered.

**Integrating system overheads into the response-time analysis.** Some system overheads are straight-forward to integrate into the RTA. The fork/join costs can be added to the computation of aggregate strands in a parallel group. Based on the number of cores a task uses, the amount of fork/join system overhead added is derived from the results of our microbenchmarks in Section IV. However, to derive a tight bound on these overheads using a measurement-based approach, we would have to run the system with all possible permutations of threads across sockets – from one thread per socket, to no socket crossings, to all cores used. We don't believe this amount of measurement is practical. Instead, the assignment algorithm in Section VII ensures that tasks only cross a socket boundary when they are assigned to all the cores on that socket. This matches how the microbenchmarks are parametrized and measured, thus enabling the use of the simple microbenchmarks measured across one varying parameter (number of cores).

*FJOS overheads.* However, there are some subtle cases that require further attention. Notably, in FJOS, there are

two areas that require additional treatment. First, the IPI for an ACAP are sent to a destination core regardless of which thread is executing on that core. Thus the “top-half” overhead of these interrupts must be accounted for at the highest priority. Our measurements have shown that imposes at most an overhead of 0.492  $\mu$ -seconds per IPI. Note that the thread that is activated by a ACAP trigger is only switched to if it is of the highest priority, so this high-priority overhead only comes from IPI reception. The rest of the overhead for thread activation, and all the activities in the worker for a fork are accounted into the activated thread. Second, the dissemination threads for the multi-cast of thread activations execute at a high priority to prevent the delay of the subsequent activations. For simplicity, here we are pessimistic: regardless of how many cores on a socket the dissemination thread will activate, we assume the worst – all 10 on our system. We measure this overhead to have a maximum of 14.10  $\mu$ -seconds.

*Linux overheads.* Linux-based implementations do not send IPIs to activate blocked threads if they do not have the highest priority on the destination core. This demonstrates a trade-off in FJOS that attempts to maintain strong separation for data-structures across cores. In an effort to avoid accessing shared data-structures between cores, some optimizations are sacrificed. However the benefits outweigh the costs both for the spin-based approaches that share only wait-free data-structures between cores, and for block-based approaches that provide a thin abstraction over IPIs and IPI multi-cast.

*Practical measurement and integration of fork/join overheads.* As a practical matter, we wish to avoid measuring the many separate aspects of fork/join. Consider the separate overheads for a “parent” strand that starts the fork, and the workers within the parallel stage. These include the overhead for the forking in the parent, the overhead in the worker component of activating, the worker’s overhead in a join, and also the parent’s overhead in a join. To take these measurements accurately is quite difficult because they require measurement synchronization that itself imposes significant overhead. It is much simpler to instead measure the end-to-end overhead of the entire fork/join operation, and integrate that into the RTA as explained above (*i.e.* essentially adding this overhead at any fork point). However, if a core is shared by multiple tasks, then accurate accounting is more difficult: we must add this end-to-end overhead both onto the parent thread *and* onto the local worker thread. Though this double-counts the fork/join overheads, it is necessary to account for the interference on other tasks on the core. Note that this is done rarely because, as we will see in the next section, the assignment algorithm attempts to prevent tasks from sharing cores.

## VII. OVERHEAD-AWARE ASSIGNMENT ALGORITHM

In this section we introduce a simple assignment algorithm to decide task priorities and to assign strands to threads on cores. Confirming the results from [7], we have found that inter-task interference has a significantly negative impact on the RTA for a fork/join task. Interference on fork/join tasks has a negative impact because its contribution from multiple

cores can easily have a disproportionate effect on the jitter of later segments. Thus, the algorithm we use is a greedy heuristic that tries to use a *minimal* number of cores to make a task schedulable. This has the effect of often devoting cores exclusively to specific tasks. Given the detrimental impact that crossing sockets has on synchronization [16] and OPENMP overheads, we use an assignment algorithm that explicitly attempts to avoid separating tasks across socket boundaries. In the evaluation section we compare against a more traditional assignment algorithm which assigns strands using worst-fit which spreads high-priority tasks across many cores.

The assignment algorithm takes these steps:

- 1) Assign all tasks priorities consistent with a deadline-monotonic policy. All strands that constitute a task share the priority of the task.
- 2) Order tasks from highest priority to lowest. We studied the effect of different task orders and found highest priority first works best. We believe this maximizes the chances of avoiding having high-priority tasks spreading their interference between many cores.
- 3) For the first unassigned task  $\tau_i$ , decide the minimal possible number of cores it requires:  $N = \lceil U_i \rceil$ .
- 4) Decide the minimal number of sockets to provide  $N$  cores:  $S = \lceil N/CPS \rceil$  where  $CPS$  is the number of cores per socket.
- 5) If  $N > Q$  (total number of cores), we conclude the assignment fails and the system is not schedulable. Otherwise, choose the  $S$  sockets with the largest available utilization. Within those sockets, choose  $N$  cores with largest available utilization and store these in an array  $cores[0..N-1]$ .
- 6) Starting from  $\tau_{i,0}$ , assign strands stage by stage. For  $\tau_{i,j}$ , assign the strands in that stage to  $cores$  using round-robin policy, *i.e.* assigning  $\tau_{i,j}^k$  to  $cores[k \bmod N]$ .
- 7) Schedulability test: after the assignment, run a schedulability test with the analysis introduced in Section VI.
- 8) If the schedulability test in the previous step fails,  $N = N + 1$  and goto step 4. Note that only when the  $N$  increases beyond the number of cores in a socket, do we assign across sockets (thus maintaining consistency with the microbenchmark overheads of Section IV). If the test succeeds, the task has been assigned; goto step 3 to attempt to assign the next task. If all tasks are assigned, the assignment is valid and the system is schedulable.

**Assignment algorithm run-time overhead.** The assignment algorithm is polynomial and its asymptotic complexity is  $O((|T| \log |T|) + (|T| \times ((Q \log Q \times SK \log SK) \times ST \times R) \times Q))$  where  $ST$  is the number of strands in a task (*i.e.* for a task  $\tau_i$ ,  $ST = \sum_{j < s_i} n_{i,j}$ ),  $SK$  is the number of sockets, and  $R$  is the complexity of performing an RTA calculation from [7]. In other words, for each task, we must inspect sockets and cores, and assign strands and do the RTA. The outer loop (step 8) executes at most  $Q$  times. Note that if overheads are measured in cycles, the worst-case cost of RTA can be quite large, though still strictly polynomial. The runtime is reasonable in practice: in our Java implementation of the assignment algorithm, with the random tests described

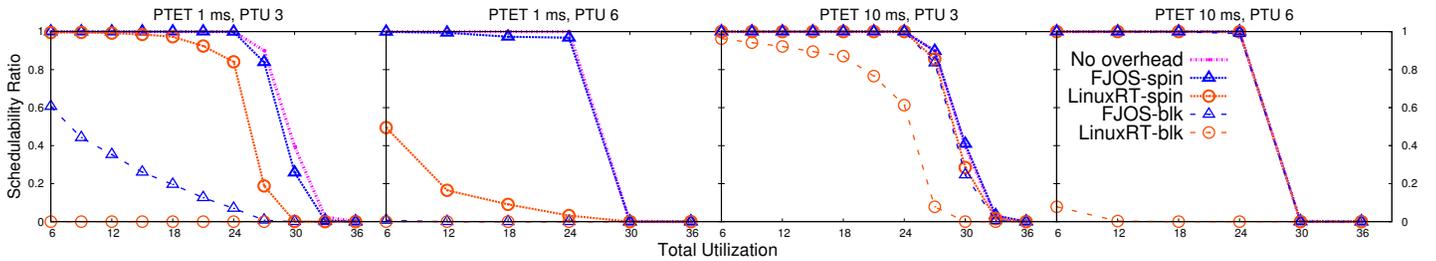


Fig. 7: Effect of system overheads for FJOS and LinuxRT with varying Per-Task Execution Time (PTET), and Per-Task Utilization (PTU).

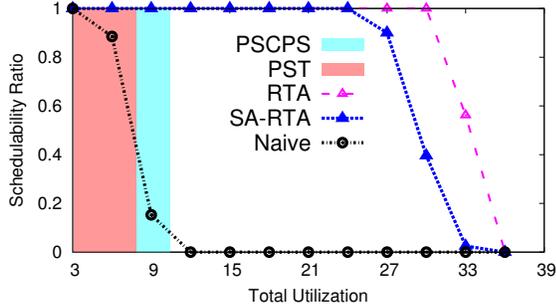


Fig. 6: Impact of assignment algorithms on schedulability.

in Section VIII, we never observed it to take more than ten seconds for 1000 random systems, and it would most-often take around a second.

## VIII. SCHEDULABILITY EVALUATION

**Fork/join system generation.** In this paper there is no constraint on system utilization or critical path length when generating random systems. We would like to study system schedulability under a wide variety of fork/join configurations. We use the following parameters to characterize fork/join parallel tasks: 1) number of strands in each parallel group, with default value range  $[1, 100]$ ; 2) number of parallel stages (separated by a minimal sequential stage) per task, which is set to 10; and 3) per task utilization, with default value range  $[2, 4]$ . All distributions are taken from a uniform distribution. Task periodicity  $p_i$  is decided by  $U_i$  and  $c_i^{max}$ , which will be studied in the evaluation. To complete a fork/join system, tasks are generated until the desired total utilization is reached. Only systems that are schedulable with infinite cores are kept. For each configuration of the simulations (*i.e.* every point in each graph), we generate 1000 random systems and report average results.

### A. Overhead-aware Assignment Algorithm

To investigate the impact of the assignment algorithm, without the impacts of system overheads, Figure 6 compares: (1) PST – the resource augmentation bound for fixed-priority scheduling in the the Parallel Synchronous Task model [6], (2) PSCPS – the resource augmentation bound of the Parallel Scheduling for Cyber-Physical Systems from [1], (3) Naive – a simple worst-fit strand assignment, (4) SA-RTA – our socket-aware assignment algorithm (Section VII) that attempts to prevent tasks from crossing socket boundaries, and (5) RTA – the same as the previous, except that it ignores socket boundaries. For the first two algorithms, any task set within their shaded areas is schedulable.

**Discussion.** We confirm previous results [7] that show that naive assignments for fork/join tasks results in disappointing schedulability. However, the RTA significantly increases in power when combined with this paper’s assignment algorithm. When compared to stronger theoretical results such as PST and PSCPS, our assignment algorithm paired with the RTA provides significantly higher, practical schedulability for random task sets.

The gap between the RTA and SA-RTA demonstrates a trade-off. To avoid taking system overhead benchmarks for all permutations of threads spread between cores, and to minimize the particularly detrimental overhead of cross-socket thread interaction, we avoid socket crossings until a task is assigned to all cores on a socket. This adds some pessimism seen here, but enables the further overhead-aware analysis without unrealistic numbers of overhead measurements for all possible thread assignments, and practically results in systems with increased schedulability.

### B. Overhead-aware Schedulability Evaluation

Figure 7 displays the effect of the system overheads of FJOS and Linux on system schedulability. The main factors that affect system schedulability with respect to system overheads are (1) system total utilization, (2) per-strand execution time (relative to system overheads), and (3) per-task utilization (PTU). The former alters the impact of overhead on schedulability, and the latter will increase system overheads as threads are spread across more cores. This requires that system overheads be derived from values for larger core numbers in Figure 4. Note that in all of these results, the “spin” versions use spin-based implementations except in one case: they use blocking when multiple tasks share a core to avoid undue interference between them. However, for the “blocking” results all implementations solely use blocking.

**Discussion.** As expected, the general trends are that it is more difficult to schedule tasks that either have a higher per-task utilization, or that have a higher ratio of task execution time to overhead. Generally, we can see that the overheads affect system schedulability in predictable ways. The only system that can schedule larger ( $PTU = 6$ ) tasks with a 1ms WCET is FJOS-spin, given its small spinning overheads. On the other hand, for smaller tasks ( $PTU = 3$ ), with an execution time of 10ms, all systems provide significant schedulability, even LinuxRT-block. Notably, FJOS-blk is as effective as LinuxRT-spin. In all cases, FJOS-spin is close to the schedulability without considering taking overheads. In summary, both FJOS implementations dominate (in our results) the comparable mechanisms in GOMP, and provide predictable

fork/join for tasks with small execution time, and with large utilizations.

*Impact of Per-Task Utilization (PTU) and Per-Task Execution Time (PTET).* Whereas Figure 7 explores the effects of changing both per-task utilization, and task size (thus indirectly modifying the strand execution time), this subsection does separate parameterizations for each of these dimensions.

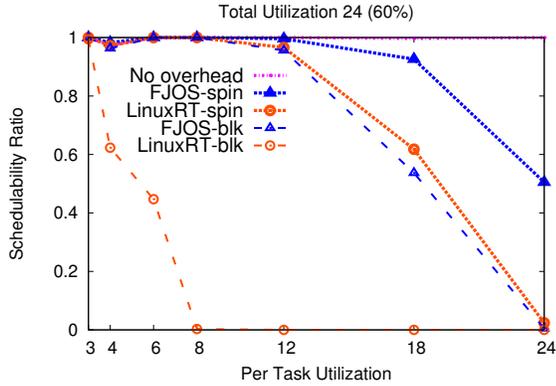


Fig. 8: System schedulability for differing per-task utilizations (PTUs) for a system with 60% total utilization.

Figure 8 plots the different implementations for differing per-task utilizations when per task execution time is 25ms. This effectively assesses how the overheads of the different implementations scale, and how that affects schedulability. Note that the measurement at  $PTU = 18$  includes one task with average size 18, and another smaller task to consume the desired 24 total utilization.

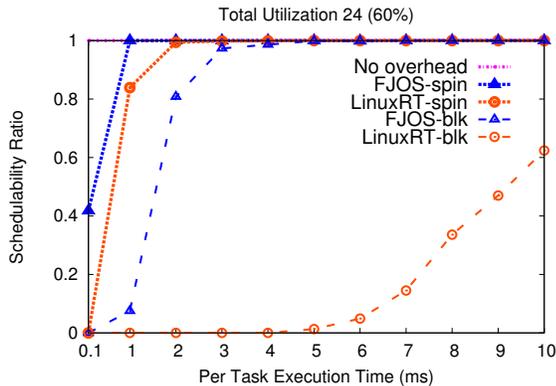


Fig. 9: System schedulability for differing task execution lengths for a system with 60% total utilization.

Figure 9 plots the fork/join implementations for differing task execution lengths when per task utilization is 3. This indirectly impacts the ratio of fork/join overhead to strand execution time. Thus, as the task execution time decreases, so does system schedulability as overheads begin to dominate.

**Discussion.** The low overhead of the FJOS-spin implementation is reflected in the system’s schedulability. Surprisingly, the FJOS-block implementation strongly tracks the LinuxRT-spin implementation (Figure 8). This is significant as it has the benefits of blocking, thus enabling other tasks to share the core, to do best-effort processing and to conserve power, while maintaining high performance and predictability.

## IX. RELATED WORK

**Fork/join.** While other fork/join analyses [1], [5], [6] provide augmentation bounds for restricted forms of fork/join parallelism, our results demonstrate that a response-time analysis paired with an optimized and predictable implementation and a system overhead-aware assignment algorithm can significantly raise system utilization. Others [8] approach the problem of dynamic work distribution for tasks with utilization less than 1 to decrease lateness for soft real-time systems.

On the implementation-focused side, [3] provide a static assignment algorithm, and an implementation of the per-core scheduler (including synchronization). Their overheads mirror that of the LinuxRT results in Section IV, and in FJOS, we provide an execution environment customized for efficient, practical fork/join computation.

We utilize the resource time analysis provided by [7], but find that it must be paired with an intelligent assignment algorithm to provide high system utilizations.

**Dependent-task frameworks.** Asynchronous frameworks that can exploit pipelined, and graph-driven parallelism via streaming between separate tasks with explicit dependencies [17], [18], [19]. This research requires the fork/join model, a much more constrained version of parallelism, but one that is popular and well-supported. Future work includes extending the efficient and predictable system support provided by FJOS to more general and unrestricted forms of parallelism.

## X. CONCLUSIONS

This paper introduces FJOS for practical, predictable, and efficient intra-task parallelism. We review an existing fork/join system and introduce the design and implementation of FJOS that is optimized for predictable, efficient fork/join computation. Results show that FJOS has significantly less overhead than GOMP for both spinning and blocking implementations – 2.5 and 5x smaller worst-case overhead at 40 cores, respectively. Surprisingly, *blocking* in FJOS is competitive with GOMP’s *spinning* implementation – despite having context switching and IPI overheads – when a task crosses socket boundaries. Schedulability analysis tests demonstrate that our assignment algorithm increases system utilization for random task sets by more than a factor of 3 over an existing RTA. When system overheads are accounted for, results show that FJOS is particularly useful either with tasks with a high ratio of the number of stages to the total execution time, or for tasks with large utilizations, thus requiring the use of many cores. We conclude that FJOS provides a strong foundation for predictable, real-time intra-task parallelism. Find the system’s and simulation’s source code at <http://composite.seas.gwu.edu/>.

*Acknowledgments.* We’d like to thank Xiang Gao for his help with the libccv results, and the anonymous reviewers that provided helpful suggestions that greatly improved the paper.

## REFERENCES

- [1] J. Kim, H. Kim, K. Lakshmanan, and R. R. Rajkumar, “Parallel scheduling for cyber-physical systems: analysis and case study on

- a self-driving car,” in *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems*, 2013, pp. 31–40.
- [2] “OpenMP: <http://www.openmp.org>, retrieved 9/21/12.”
  - [3] D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C. Gill, and C. Lu, “A real-time scheduling service for parallel tasks,” in *Proceedings of the 2013 19th IEEE Symposium on Real-Time and Embedded Technology and Applications*, 2013.
  - [4] A. J. Dorta, C. Rodriguez, F. d. Sande, and A. Gonzalez-Escribano, “The openmp source code repository,” in *the Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2005.
  - [5] K. Lakshmanan, S. Kato, and R. R. Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, 2010.
  - [6] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, “Multi-core real-time scheduling for generalized parallel task models,” in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2011.
  - [7] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig, “Response-time analysis of parallel fork-join workloads with real-time constraints,” in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2013, pp. 215–224.
  - [8] L. Nogueira and L. M. Pinho, “Server-based scheduling of parallel real-time tasks,” in *Proceedings of the tenth ACM international conference on Embedded software*, ser. EMSOFT ’12, 2012.
  - [9] 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.
  - [10] G. Parmer and R. West, “Predictable interrupt management and scheduling in the Composite component-based system,” in *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2008.
  - [11] 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)*, 2010.
  - [12] 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.
  - [13] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz, “Predictable interrupt management for real time kernels over conventional pc hardware,” in *RTAS ’06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 14–23.
  - [14] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schpbach, and A. Singhanian, “The multikernel: A new OS architecture for scalable multicore systems,” in *Symposium on Operating System Principles (SOSP)*, 2009.
  - [15] Q. Wang, J. Song, G. Parmer, J. Wittrock, Y. Y. Wu, and T. Hossain, “Hijack<sub>linux</sub><sup>cos</sup>: Toward practical, predictable, and efficient OS co-location using Linux,” in *Real-Time Linux Workshop*, 2012.
  - [16] T. David, R. Guerraoui, and V. Trigonakis, “Everything you always wanted to know about synchronization but were afraid to ask,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP ’13. New York, NY, USA: ACM, 2013, pp. 33–48.
  - [17] C. Liu and J. Anderson, “Supporting graph-based real-time applications in distributed systems,” in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011.
  - [18] P. Jayachandran and T. Abdelzاهر, “Reduction-based schedulability analysis of distributed systems with cycles in the task graph,” *Real-Time Syst.*, 2010.
  - [19] K. Tindell and J. Clark, “Holistic schedulability analysis for distributed hard real-time systems,” *Journal of Microprocessing and Microprogramming - Parallel processing in embedded real-time systems*, 1994.

## APPENDIX

### A. COMPOSITE Background and FJOS Details.

**Scheduling in COMPOSITE.** Scheduling policies are defined in user-level components in COMPOSITE [10]. COMPOSITE attempts to avoid the overhead of switching to the scheduler protection domain for interrupts, including IPIs. When a thread that subsequently blocked on `acap_wait` is activated by an IPI from `acap_trigger`, the interrupt handler routine parses structured tables shared between the scheduler and

the kernel, and determines if the currently active thread has a lower priority than the activated thread. If so, the thread is context switched without invoking the scheduler. The scheduler must fill in these tables appropriately given its own policies. When a thread calls `acap_wait` on a capability for an event that has not been triggered, it must block. If no other threads have been woken up, or activated since it was activated, the system knows it can directly switch back to the thread it preempted, thus avoiding a scheduler invocation. Otherwise, the thread is made to invoke the scheduler component. The scheduler will then update its structures, and block the thread (*i.e.* remove it from the runqueue). For more details on how interrupts interact with user-level schedulers in COMPOSITE, see [10].

**Inter-component fork/join.** COMPOSITE focuses on enabling the encapsulation of a component’s functionality behind its interface, thus enabling multiple, competing implementations. If one of a specific implementation uses OPENMP parallelism, and it is invoked by other components that also use OPENMP, then nested parallelism is essentially required. The benefit of this form of inter-component fork/join in COMPOSITE is that any component (ranging from image processing services, all the way to memory managers) can use OPENMP. The PARMGR transparently supports this inter-component fork/join by tracking only the fork/join structure of the system, and not the specific components. A global policy that statically determines the allocation of computation to cores for different segments must still guide the PARMGR’s creation of ACAPs and threads, and a global analysis of the system’s timeliness is still required.