Callisto: Co-Scheduling Parallel Runtime Systems

Tim Harris
Oracle Labs, Cambridge, UK
timothy.l.harris@oracle.com

Martin Maas
UC Berkeley, USA
maas@eecs.berkeley.edu

Virendra J. Marathe
Oracle Labs, Burlington, USA
virendra.marathe@oracle.com

Abstract

It is increasingly important for parallel applications to run together on the same machine. However, current performance is often poor: programs do not adapt well to dynamically varying numbers of cores, and the CPU time received by concurrent jobs can differ drastically. This paper introduces Callisto, a resource management layer for parallel runtime systems. We describe Callisto and the implementation of two Callisto-enabled runtime systems—one for OpenMP, and another for a task-parallel programming model. We show how Callisto eliminates almost all of the scheduler-related interference between concurrent jobs, while still allowing jobs to claim otherwise-idle cores. We use examples from two recent graph analytics projects and from SPEC OMP.

1. Introduction

It is increasingly important for multiple parallel applications to run well together on the same machine. There are several trends: (*i*) the need to make effective use of multi-core hardware leads to increasing use of parallelism within software, (*ii*) the desire to use hardware efficiently leads to greater co-location of workloads on the same machine, and (*iii*) parallel applications are expected to "just work" without careful tuning to specific systems.

Currently, parallel runtime systems interact poorly with the schedulers used by operating systems and virtual machine monitors. We see three problems:

First, preemption occurs at inconvenient times. A classic example is while holding a lock: threads needing the lock cannot proceed until the lock holder runs. Another example is a parallel loop in which threads claim batches of iterations: the loop cannot terminate if a thread is preempted mid-batch.

Second, when a thread waits, it must decide whether to spin or to block. The best decision depends on information that is not usually available to the thread (e.g., which other threads are running) [4, 15, 16, 36]. There can be a tension between a process's own performance and system-level concerns (e.g., using cores productively instead of for spinning).

Finally, when multiple jobs run together, the CPU time that each receives can be drastically different and hard to control. It can depend, for example, on whether the OS prioritizes threads that have recently unblocked.

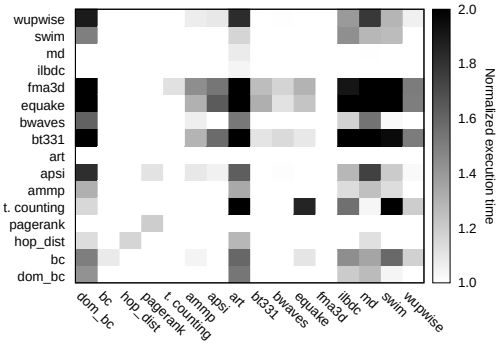


Fig. 1. Execution time for jobs running together on a 2-socket machine and competing for access to 32 h/w contexts. Each square shows the execution time of the job on the y-axis when running with the job on the x-axis. We normalize to the job's execution time running alone on 1-socket.

Fig. 1 illustrates these problems. We return to the details in Section 4 but, in outline, the experiment examines the performance of pairs of benchmarks sharing a 2-socket x64 Linux machine. We ran each pair of benchmarks together. For each combination, we repeated each benchmark until both of them have run at least 5 times. We then plotted the median execution time of the benchmark on the y-axis when sharing the 2-socket machine with the benchmark on the x-axis, normalized to how long the benchmark took in isolation on just 1-socket. Darker squares indicate progressively slower performance. White squares indicate either a speed-up or at least no slow-down. Ideally, with two jobs sharing two sockets, we would expect the complete heat map to be white. However, in practice, some jobs take up to 3.5x longer when sharing the machine.

In this paper we introduce Callisto, a resource management layer for parallel runtime systems. Compared with the

EuroSys '14, April 13–16, 2014, Amsterdam, Netherlands..

http://dx.doi.org/10.1145/2592798.2592807

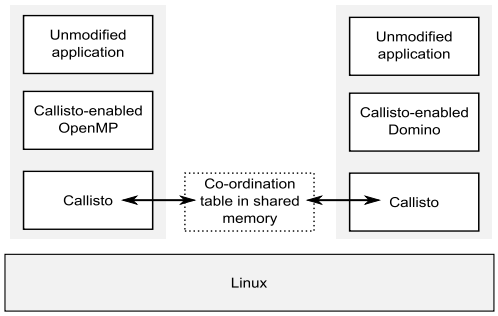


Fig. 2. The structure of a system using Callisto.

results in Fig. 1, Callisto lets us reduce the worst interference to 1.25x execution time. We use three techniques:

- Multiple parallel jobs coordinate their resource demands to leave exactly one runnable s/w thread for each h/w context (Section 2.1). This generally avoids the OS preempting threads transparently.
- We provide CPU time to runtime systems using an up-call mechanism, inspired by scheduler activations [1] (Section 2.2). A runtime system divides its work into small pieces which generally run to completion within each of these up-calls, and which can be multiplexed over however many h/w contexts are available (Section 2.3).
- We provide a single API for building the synchronization primitives exposed to applications (e.g., locks), for the synchronization within a runtime system (e.g., managing task pools), and for synchronization within Callisto itself (Section 2.4). By combining these three kinds of synchronization, we expose to Callisto information about which work is ready to execute, and we provide a single point to make spin/block decisions.

Fig. 2 shows the overall structure of our prototype implementation. In this prototype, Callisto is a user-mode shared library which links with modified versions of different parallel runtime systems. Multiple instances of the library interact through shared memory to cooperatively control the use of the h/w contexts. Runtime systems need to be adapted to use Callisto. However, applications built over these runtime systems operate unmodified, and there are no OS changes. Our cooperative approach using a shared library is intended to allow experimentation, rather than necessarily being how Callisto would be deployed in practice (e.g., when it would need to protect against failures, bugs, or malicious behavior in the applications it controls). We discuss different deployment options in more detail at the end of this paper.

Section 3 discusses two Callisto-enabled runtime systems: OpenMP, and Domino [12] (a fine-grained task-parallel system). We show how unmodified OpenMP applications can run over a dynamically varying number of h/w contexts while still meeting requirements in the OpenMP API for how pieces of work map to specific threads.

Section 4 presents initial performance results from a Linux system with 32 h/w contexts. When running a single job, Callisto varies between a 22% slow down and a 22% speed up (mean slow down of 3.1%). When running pairs of jobs, we show that Callisto (*i*) reduces the likelihood of interference, (*ii*) reduces the severity of interference, and (*iii*) increases the ability for peaks in one job's CPU demand to benefit from troughs in another's.

Section 5 describes related work, and Section 6 covers limitations of our approach, along with future work, and conclusions.

2. Callisto

In this section we introduce the assumptions made by Callisto, and the kinds of workload we target. We then introduce the main techniques we use—dynamic spatial scheduling (Section 2.1), an up-call interface for passing control to the runtime system (Section 2.2), a mechanism for the runtime system to express the parallel work it has available (Section 2.3) and a unified synchronization mechanism for waiting (Section 2.4).

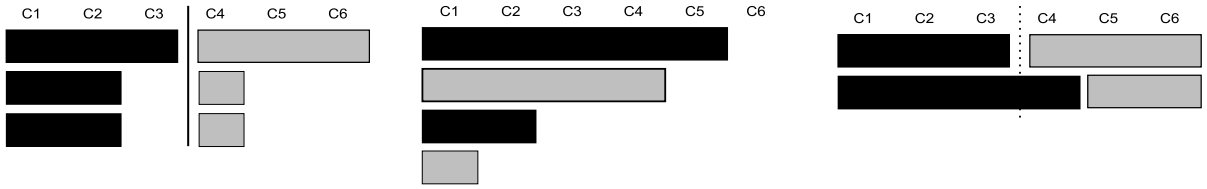
Setting and assumptions. We focus specifically on improving co-scheduling between jobs using abstractions such as CDDP [12], Cilk [11], Data-Parallel Haskell [30], Galois [17], and OpenMP. In these settings, workloads identify sources of parallelism without explicitly forking threads. The number of h/w contexts used is usually set either by parameters, or by the runtime system.

We assume that parallel sections of jobs are generally CPU-bound and so each s/w thread is able to completely use a single h/w context. Some parallel runtime systems, such as Grappa [25], use lightweight threads to issue IO operations concurrently; these systems also fit our model, multiplexing lightweight threads over a single s/w thread, and combining IO operations into batches.

We do not currently focus on distributed map-reduce-style frameworks. In those settings, the processes themselves are typically independent, and are dispatched to machines under the control of a central scheduler. Systems such as Mesos [13] and Omega [32] have been used to dispatch processes from different jobs to a shared set of machines in a cluster. However, within a single machine, multiplexing the resulting set of processes seems to be handled relatively well by existing OS schedulers.

2.1 Dynamic Spatial Scheduling

Our first idea is to ensure that a set of jobs leaves exactly one runnable s/w thread pinned to each h/w context. Runtime systems claim h/w contexts via a table in shared memory, adjusting their own demands when other jobs start or finish, and when the amount of available work changes. Our current implementation assumes cooperative runtime systems (we discuss the merits of this at the end of the paper).



(a) Static spatial allocation (b) Gang-scheduled allocation (c) Dynamic spatial allocation

Fig. 3. Contrasting different policies for allocating 6 cores to two jobs. Time quanta run top to bottom.

We dynamically partition the h/w contexts between jobs. Each job has a fixed portion of the CPUs when all of the jobs are runnable, but can make use of otherwise-idle h/w contexts when available. This approach has two benefits over statically partitioning a machine (Fig. 3a), or gang-scheduling jobs [26] in time slices (Fig. 3b): First, we can improve utilization if peaks in one job's demands coincide with troughs in another's. Second, jobs usually get less benefit from each additional h/w context: therefore it is better to split h/w contexts among jobs if we can do so without generating interference.

Our scheduler is controlled by two policies. First, a spatial scheduling policy defines which jobs should use which h/w contexts when all the jobs are runnable. The spatial scheduling policy grants a single job high priority for each h/w context. This defines a state for the system when jobs remain CPU-bound, and lets us explore placement policies such as whether to allocate all of the h/w contexts in a given core to the same job. We repeat spatial scheduling when jobs start or terminate.

Second, a temporal scheduling policy defines how to use a h/w context when its high priority job is idle. There is a trade-off between re-allocating spare resources, versus letting the high priority job keep the h/w context in the hope that it will have new work soon (or simply to avoid disruption to the state of local caches). Our approach is:

1. Each job has a dedicated h/w context for its "main" thread. In OpenMP, this thread is responsible for spawning parallel work and the performance of the job is highly dependent on the main thread's execution.
2. If a job runs out of work, it retains the h/w contexts on which it has high priority for a configurable hysteresis delay H_{high}. This rule lets the job retain resources if it has short sequential sections in between parallel work.
3. After this delay, h/w contexts are re-allocated to other jobs. If these, in turn, run out of work then they can retain the h/w context for a delay H_{low} (assuming that the job with high priority is still out of work, and there are no other low priority jobs with work). After H_{low} without work, the h/w context is returned to the high priority job. The rule avoids low priority jobs losing h/w contexts during short sequential sections, while returning the allocation to a "clean" state if no job has any work.
4. If a job is running on a h/w context on which it does not have high priority, then it must periodically check if the context's high priority job has become runnable, and yield if requested. The check-in period is P_{low}. This rule lets a job regain access to its high priority h/w contexts.
5. If a job has high priority on a h/w context then it must still periodically check for changes to the spatial allocation of contexts. The check-in period is P_{high}.

Implementation. The check-in protocol is cooperative. A runtime system is expected to check-in at times when preemption would be convenient (e.g., between batches of loop iterations, rather than within a batch). The check-in tests are made by the runtime system, without changes to application code. We add a periodic "runaway timer" to force a check-in if one does not occur within the required interval—this is currently done in user mode within the runtime system itself, but could be moved to the kernel to enforce check-ins if necessary. By default we set the two hysteresis delays to 10ms, P_{high} to 100ms, and P_{low} to 1ms.

Callisto maintains a table in shared memory holding per-h/w-context information: (*i*) which job has high priority for that context, (*ii*) which job is currently running on the context, (*iii*) context-local timestamp values for when the next check-in is due, and when the hysteresis delay ends, (*iv*) per-job flags indicating if that job wishes to run on the context, and (*v*) a pthread mutex and condition variable used by jobs to block/wake-up when passing the core to one another.

In the common case, a check-in test simply compares the current time against the time at which the next check-in is due. When the time is reached, the full test checks which job should run on the context (a deterministic function of the shared state), and yields if necessary. To switch jobs, the yielding job signals others jobs waiting on the condition variable, before blocking.

2.2 Interface to Callisto-Enabled Runtime Systems

A runtime system on Callisto operates by dispatching work to run over a set of *workers*. A job is expressed as a set of *work tickets*, each of which represents a source of parallelism—e.g., a complete parallel OpenMP loop would

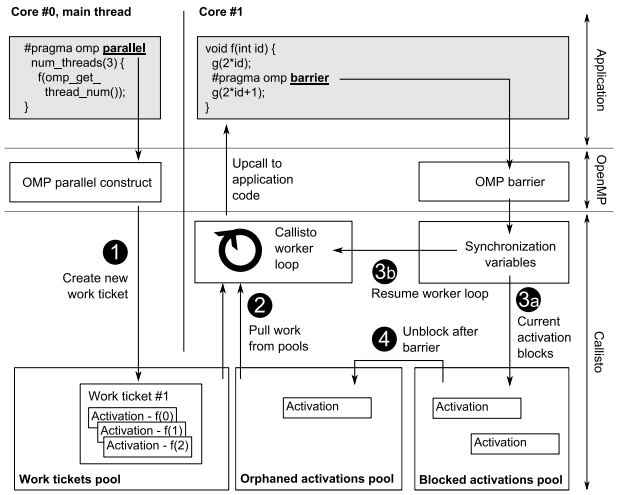


Fig. 4. System structure with an OpenMP loop which uses three threads to call the function `f`.

be represented by a single work ticket. Whenever a worker is allocated a h/w context, it executes parts of a work ticket by making an up-call to the runtime system at a fixed entry point. The code at that entry point then executes part of the work (e.g., a batch of loop iterations). The use of up-calls is inspired by scheduler activations [1] and so we call the entry point the runtime system's activation handler, and we call each up-call an *activation*.

Figure 4 provides an initial OpenMP example in which the main thread enters a parallel section. This uses three threads ("`num_threads(3)`"), each of which calls `f` with its own thread ID. The `f` function uses a barrier internally. With Callisto, this example is implemented by the OpenMP runtime system creating a work ticket for the parallel region (Step 1) which can be activated three times for the calls `f(0)`...`f(2)`. A worker picks up one of these calls (Step 2), allocates a stack to use, and then executes the call until it blocks at the barrier. When blocking, the partially-executed piece of work is put into a pool of blocked activations (Step 3a), and the same worker thread can switch to a fresh stack and pick up new work instead (Step 3b). Once all of the calls to `f` have reached the barrier, the blocked activations can be made eligible to run again (Step 4) and be picked up by a Callisto worker and completed.

We now describe the interfaces for managing work tickets (Section 2.3), and then the interfaces for building barriers and other synchronization primitives (Section 2.4).

2.3 Managing Work Tickets

To illustrate how parallel work is managed over Callisto, we use the following OpenMP loop as a larger running example:

```
#pragma omp parallel for schedule(dynamic,100)
for (int i = 0; i < 1000000; i++) {
  arr[i] *= 2;
}
printf("Done\n");
```

```
// Start-of-day
Worker[] CreateWorkers();
// Managing work
uint64_t CreateWorkTicket(void *data,
            int max_concurrency);
void SetTicketDrained(uint64_t ticket);
void WaitTicketDrained(uint64_t ticket);
void WaitTicketComplete(uint64_t ticket);
```

(a) Operations exposed by Callisto.

```
void Activate(Worker *w, void *data,
            uint64_t ticket);
```

(b) Operations implemented by a Callisto-enabled runtime system.

```
bool ShouldYield();
void YieldPoint();
```

(c) Operations for coordination between jobs.

Fig. 5. APIs used for creating and managing work.

This loop iterates over a large array (`arr`), doubling each element. The pragma indicates that the iterations can run in parallel, and that threads should share the work by dynamically claiming batches of 100 iterations. "Done" is printed only after all of the iterations have completed. An OpenMP compiler usually extracts the body of the loop into a separate function, and creates a data structure holding the bounds of the loop and a shared counter to indicate the next batch of iterations to be claimed. Threads use an atomic increment on the shared counter to claim a batch of iterations, run those, and then return for more.

Workers. Conceptually, every job has one worker for each h/w context in the machine. The number of workers that are actually running will vary over time under the dynamic spatial scheduling policy. Fixing workers to h/w contexts, rather than having them migrate within a machine, enables the runtime system to cache information about the physical structure of the machine (e.g., which workers are co-located on the same core), and use that information to build data structures such as SNZI trees [9]. Also, although our current implementation supports only homogeneous systems, we believe it will be easier to extend the fixed-worker model to support heterogeneous systems in the future.

Fig. 5 shows the API used for managing work. The `CreateWorkers` function is used at start-of-day to initialize a job's workers. Concretely, each worker is implemented by a pthread pinned to the associated h/w context, and blocked on the context's condition variable when it should not receive the CPU. Each worker is in one of two states: *waiting* (created, but either waiting for work to run, or waiting for synchronization within its current work), or *runnable* (with work to execute, whether or not it is actually running on its h/w context). Workers start in the *waiting*

state, and so simply creating a set of workers does not cause any computation to happen.

Work tickets. A *work ticket* represents a source of parallel work. `CreateWorkTicket` takes two parameters: (*i*) an opaque data pointer which is passed back to the activation handler when running this ticket, and (*ii*) a bound on the maximum number of concurrent activations that should occur (and hence on the amount of memory needed for their stacks). In the OpenMP example, this maximum comes from the `OMP_NUM_THREADS` environment variable which OpenMP uses to set the number of s/w threads used.

A work ticket is in one of three states: *active* (created, and able to run), *drained* (all of the work has been started, so further activations are unnecessary), or *complete* (all of the work has finished, rather than simply being in progress). In our example, `SetTicketDrained` is called when the final batch of loop iterations starts. The main thread calls `WaitTicketComplete` before printing "Done".

Activations. An *activation* executes part of a work ticket. One of Callisto's workers makes an up-call to the runtime system's `Activate` function (Fig. 5b), identifying the worker that the up-call starts on, the pointer from the ticket, and the ID of the ticket itself. In our example, the pointer identifies an OpenMP structure describing the parallel loop.

Activations are intended to be short-lived and to cooperate with the spatial scheduling algorithm: when invoked, the activation handler should perform work in the runtime system and check in periodically by calling `ShouldYield` (Fig. 5c). When `ShouldYield` returns true, the activation should return from its up-call. Usually, activations are naturally short-lived when executing parts of loops, or small tasks from a work-pool. However, there are two cases where an activation might not be short-lived:

First, application code may block—e.g., an OpenMP parallel region may contain a barrier in the middle of it. If this happens then the activation will be resumed once it unblocks (possibly on a different worker). We call these *orphaned activations*, and workers run them in preference to starting new activations. To let these long-running activations move between workers, each up-call runs on its own stack, independent from the one used within Callisto.

Second, application code may simply run for a long time, either intentionally or due to a bug. To avoid uncontrolled preemption, an activation can call `YieldPoint` at places where it would be convenient to pause its execution (e.g., just after releasing a lock). Internally, `YieldPoint` tests `ShouldYield` and, if requested to yield, the current activation is suspended and added to the job's set of orphaned activations. If the activation fails to check in sufficiently frequently then the runaway timer mentioned in the previous section will force it to yield in any case.

Miscellaneous functions. We provide worker-local and activation-local state. The first is used for state that is fixed to a given context (e.g., a work pool for a specific NUMA domain). Activation-local state is for information associated with a s/w thread in the programming model—e.g., it is used for the current OpenMP thread ID. It must follow the activation if it is moved to a different worker.

```
// Latches
void LatchInit(Latch *l);
void LatchAcquire(Latch *l);
void LatchRelease(Latch *l);

// Synchronization variables (SVars)
void SVarInit(SVar *s, int v);
int SVarRead(SVar *s);
void SVarWrite(SVar *s, int v);

// Blocking
typedef bool (*Sync_fn)(void *data);
void SyncWaitUntilTrue(Latch *l,
                       Sync_fn *fn,
                       void *data);

// Control over activations
void SuspendAct(act **ap);
void ResumeAct(act *a);
```

Fig. 6. Synchronization API exposed by Callisto.

2.4 Synchronization and Blocking

The second part of our API is for synchronization. It is a low-level API for use within a runtime system to build abstractions for use by application programmers (e.g., mutexes and barriers). It is also used within Callisto itself in the functions which manage synchronization on work tickets and activations. Using a common abstraction across these different levels means that spin/block decisions can be made consistently—for instance, spinning in the absence of other work to execute, or yielding the h/w context to another job.

The low-level synchronization API is not intended for use within applications. Consequently, design decisions are taken to optimize performance rather than for ease of programming. Figure 6 shows the API itself. It provides two abstractions:

Latches. In Callisto, a *latch* is a mutual exclusion lock which is used to protect other synchronization data structures (such as a full/empty flag for a work pool). Latches are never held when waiting.

Synchronization variables (SVars). An SVar encapsulates a single integer value, with read and write functions. Each SVar must be protected consistently by a latch: the latch must always be held when updating the SVar (the programmer must ensure this).

Blocking is done by calling `SyncWaitUntilTrue` with a predicate over SVars which will be true when it is possible to continue. The result of this predicate must depend only on the contents of SVars protected by the latch passed to the waiting function. Furthermore, the predicate must be written carefully so that it can be "probed" without acquiring the

latch—it must not loop or crash if it sees an inconsistent set of values in the SVars. The latch must be held before calling `SyncWaitUntilTrue`, and it will be re-acquired by the implementation before returning.

Examples. Our OpenMP barriers are implemented using an integer counter which is atomically decremented with fetch-and-add on arrival at the barrier, counting down to zero when all of the OpenMP threads have arrived. The last thread to arrive briefly acquires a per-barrier latch and increments a per-barrier generation number held in an SVar. If an OpenMP thread is not the last to arrive then it blocks, waiting for a change to the generation number.

Our OpenMP-level mutexes are implemented using an MCS-style list of per-OpenMP-thread queue nodes [24]. The lists are constructed using atomic compare and swap. Each queue node holds a latch and a single SVar. An OpenMP thread blocking on a mutex sets the SVar to 0 before calling `SyncWaitUntilTrue` to wait for the SVar to become 1.

Implementation. The `SyncWaitUntilTrue` abstraction provides flexibility to use a combination of different implementation techniques. In doing so, the aim is to provide a unified place at which spin/block decisions can be made, taking into account synchronization within the runtime system (e.g., at an OpenMP barrier) and synchronization within Callisto (e.g., waiting for a work ticket to be complete). The predicate can be evaluated either by the waiter (spinning until it is true), or the predicate can be held in a queue attached to a latch and re-evaluated whenever an update is made to one of the SVars protected by that latch.

Concretely, we implement a latch as an integer version number and a linked list of wait-queue entry structures. A latch is unlocked iff its version number is even. The `LatchAcquire` function spins until the version number is even, before using atomic compare-and-swap to increment it, making it odd. The latch protects a chain of `WaitQ` structures which hold the predicates on which code is waiting. `LatchRelease` processes the queue (we describe how this is done below), before incrementing the version number to release the latch.

`SyncWaitUntilTrue` can behave in two ways:

Active — `SyncWaitUntilTrue(l,fn,d)` starts by testing `fn(d)`. If true, it returns immediately. If false, it releases the latch and spins until the latch has been locked and unlocked at least once. (Since `fn(d)` depends only on SVars protected by the latch, the predicate's value can change only after the lock has been held. We assume that watching the single version number is faster than repeatedly probing the predicate). After observing a change, `fn(d)` is probed and, if true, the latch is re-acquired, `fn(d)` tested once again and, if true, `SyncWaitUntilTrue` returns. Otherwise, the function repeats.

Passive — In this implementation, responsibility for wake-ups is passed to the `LatchRelease` function. Calling `SyncWaitUntilTrue` checks that the predicate is false. If so, it initializes a stack-allocated `WaitQ` structure, and then the activation yields (calling `SuspendAct` to store a handle for the current activation in the `WaitQ`). When resumed, `SyncWaitUntilTrue` re-acquires the latch, checks the predicate, and returns if it is true. The `LatchRelease` function is responsible for waking activations, testing the predicates in the queue if it is nonempty. For any predicates that are true, it removes the queue entry, and calls `ResumeAct` to add the activation to the pool of orphaned activations for execution by the job's workers.

Our implementation combines these two implementations in a conventional spin-then-block approach. We initially use the active implementation until (*i*) the worker is requested to yield to another job, or (*ii*) there is an orphaned activation available to run, or (*iii*) a configurable spinning limit is reached (we use 100k cycles, but our results do not seem sensitive to the exact value chosen).

Optimizations. We applied several optimizations to the basic implementation described above: While holding a latch, we maintain a flag recording whether or not any SVars have been updated; we only consider waking threads if writes have been made. We provide specialized version of `SyncWait...` functions for common cases—e.g., waiting until a single SVar holds a specific value, or when the caller can guarantee that the predicate is false initially.

In addition to these successful optimizations, we considered a "hand-off" wake-up policy in which at most one activation would be woken per release of the latch. In turn, that first activation would be responsible for handing off the wake-up to a second wait queue entry after it has released the latch. The hope was to avoid multiple activations being woken and stampeding for the latch (since they must all acquire it before making progress). The implementation became complex because of the need to avoid forgetting to make the hand-offs; it did not lead to a performance benefit for the workloads we studied.

3. Building Runtime Systems over Callisto

In this section we describe the two prototype Callisto-enabled runtime systems—an implementation of OpenMP (Section 3.1), and a task-parallel framework (Section 3.2).

3.1 OpenMP Runtime System

Our OpenMP implementation is based on GOMP in GCC 4.8.0. We handle C/C++ and Fortran, and all OpenMP 3.0 features except for tasks, the `ORDERED` directive, and nested parallel sections. We do not believe there are fundamental difficulties in adding these; indeed, our second runtime system includes a work-pool-based implementation of tasks (Section 3.2). We briefly describe two challenges that we addressed: (*i*) avoiding unnecessary barriers, and (*ii*) reducing load imbalance in statically-scheduled loops. We then summarize implementation details. We do not require any changes to the compiler, or to the OpenMP applications.

Avoiding unnecessary barrier synchronization. The first problem we address is the use of barriers at the end of loops. Consider the following example:

```
#pragma omp parallel for
for (int i = 0; i < 1000000; i++) { ... }
#pragma omp parallel for
for (int i = 0; i < 1000000; i++) { ... }
```

Iterations from the second loop must not start until the first loop is complete. This is usually enforced with a process-wide barrier—a thread must participate in the barrier even if it has not executed any loop iterations. That can happen if the thread was not scheduled on a h/w context between the time that the loop started and terminated.

We address this problem by decoupling the notion of OpenMP threads from the specific workers that happen to execute pieces of code. Each OpenMP parallel section maps onto a Callisto work ticket, with the maximum concurrency set to the number of OpenMP threads to use. Each activation selects the next OpenMP thread, sets the activation-local storage of the current s/w thread to that of the OpenMP thread, and executes the iterations assigned to that thread. Multiple OpenMP threads can therefore be multiplexed over a single s/w thread. Switching between OpenMP threads occurs in user-mode, typically by a worker starting new activations when earlier ones block at the barrier.

Reducing load imbalance. The second challenge is that OpenMP exposes the number of threads in use to the application, and the OpenMP API provides rules about how threads are assigned work; it would be incorrect for an implementation to vary this in an ad-hoc manner. Furthermore, we must avoid introducing load imbalance. Consider this example:

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < 1000000; i++) { ... }
```

The `static` clause indicates that the loop should be divided between threads into equal batches. Static scheduling is common because it has low overheads in many implementations; it is appropriate when the number of threads is fixed, and the work in each loop iteration is constant. The problem here is that multiplexing OpenMP threads over a smaller number of h/w contexts can cause load imbalance: suppose (in an extreme case) that a loop is statically scheduled over 32 s/w threads, but that only 31 h/w contexts are available. Without care, one h/w context will execute the work from 2 s/w threads in series, doubling the completion time.

Although it is tempting to replace static scheduling with dynamic scheduling, we do not do so because (*i*) we would need to modify the compiler or application, and wished to avoid doing so, and (*ii*) OpenMP dictates cases where iterations in statically scheduled loops must run in *identical* threads between different loops. (Informally, loops must dispatch the same iterations to the same threads. This lets each thread retain local state for the iterations that it handles.)

We improve load balancing by *over-subscribing* the system with more OpenMP threads than h/w contexts. Static loops are split between this larger pool of OpenMP threads. These threads will be executed dynamically by Callisto workers based on the number of workers running and the duration of each batch of iterations. In effect, over-subscription changes a statically scheduled loop into a relatively coarse-grained dynamically scheduled one. As our results show (Section 4), multiplexing large numbers of OpenMP threads over Callisto is generally faster than using full OS threads. Over-subscription increases the parallel slack [34].

3.2 Domino Runtime System

We briefly describe a second Callisto-enabled runtime system ("Domino") providing an implementation of a task-based programming model. This model is based on parallel execution of fine-grained tasks, each typically performing a few memory reads and writes, and running to completion without any synchronization. Domino is based on our existing implementation of the CDDP programming model [12] in which tasks are spawned when an existing task writes to a memory location with a "trigger" attached to it. Constraints can be used to defer the execution of some tasks.

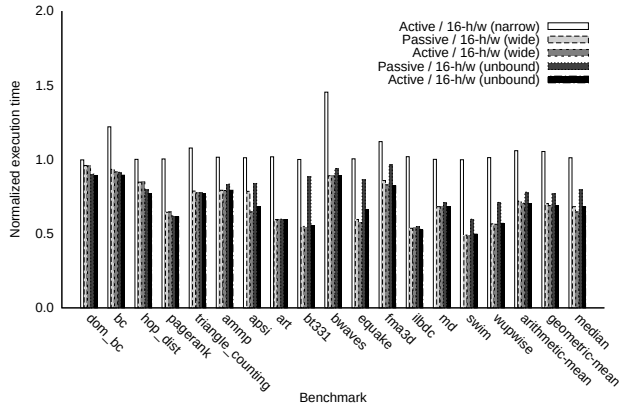
The original Domino implementation uses a fixed number of worker threads, each with a `DominoRTS` structure holding a work queue, and a per-thread set of tasks whose execution is deferred. If a thread's own queue is empty then it can steal from another's. When all work queues are empty, then items in the deferred sets are promoted to the queues. The original runtime system is not designed for use on shared machines: threads spin continually while waiting for work. Other parallel runtime systems behave in a similar way (e.g., we observed similar behavior in a version of Galois [17]).

The Callisto-enabled implementation is simplistic. It starts by allocating one `DominoRTS` per h/w context. These are held in a global list. A single work ticket represents the entire execution of the parallel computation. When activated, the Domino runtime system claims a `DominoRTS` data structure from the global list. It then executes using the work pool from that structure, stealing work from other structures as required. It calls `ShouldYield` between tasks, and returns from the activation handler if requested (releasing the `DominoRTS` to the global pool).

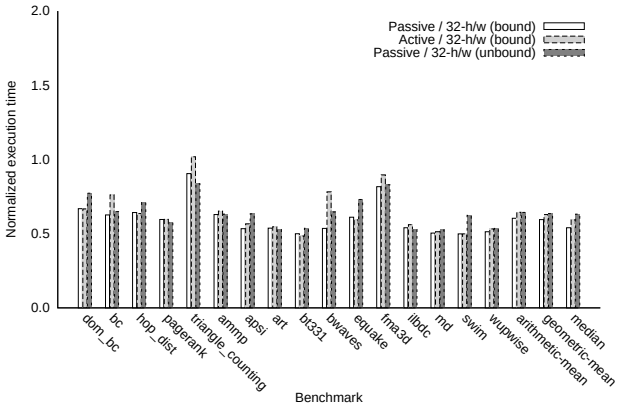
This design means that a Domino job is responsive to other jobs' demands on the machine, but the job itself will never actually block, even if it is short of work. This is not ideal from a system-wide point of view, but it provides us with an example of an aggressive runtime system.

4. Evaluation

We use 2-socket machines with Xeon E5-2660 processors at 2.20GHz (8 cores per socket, and 2 h/w contexts per core, for a total of 32 h/w contexts). Each machine has 256GB physical memory. We use Linux 2.6.32, and GCC 4.8.0.

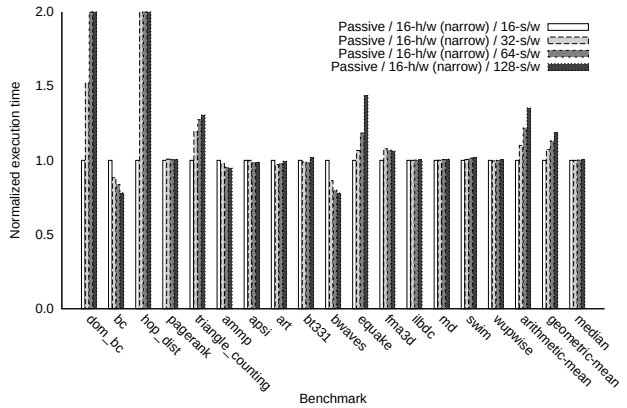


(a) 16 h/w contexts

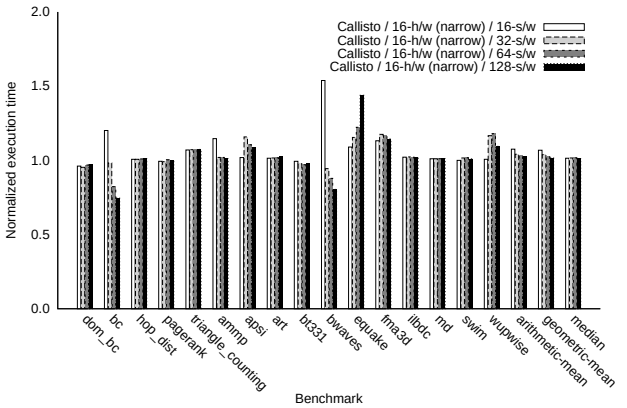


(b) 32 h/w contexts

Fig. 7. Application performance with active/passive synchronization and bound/unbound s/w threads. We use the original runtime systems, and normalize to execution using passive synchronization and all 16 h/w contexts on a single socket.



(a) Without Callisto (truncating the slowest results at 2.0x)



(b) With Callisto

Fig. 8. Varying the number of s/w threads with the original runtime systems, and with Callisto.

We use three sets of workloads. First, a Domino implementation of betweenness-centrality (BC) using our non-Callisto implementation as a baseline [12]. Second, graph analytic workloads from the public release of Green-Marl [14]: a further implementation of BC, a single-source shortest paths algorithm (hop_dist), PageRank, and a triangle counting algorithm. Green-Marl compiles these programs to OpenMP. Finally, we use benchmarks from SPEC OMP 2001 and 2012. We include all the programs supported by our OpenMP implementation. We use the 2012 version where similar benchmarks are in both suites. We set the input parameters so that each run is 10-100s in isolation on a whole machine: this means using the "large" versions of the 2001 benchmarks, and reducing the input size of some 2012 benchmarks. We show the median of 5 runs. We omit error bars because results are stable when running alone.

4.1 Single job

We start with jobs running alone, examining how well they perform without Callisto, and then compare the Callisto-enabled versions with the originals. For consistency, we normalize all results to runs where a job uses 16 s/w threads, bound to all 16 h/w contexts in a single socket. We use passive synchronization on the OpenMP runs. (Intuitively, when we turn to pairs of jobs, this 1-socket performance is the minimum we would expect when a pair shares 2 sockets.)

First, we compare the original runtime systems with active/passive synchronization, and bound/unbound threads. We say threads are bound "narrowly" when they are packed onto the h/w contexts on a single socket (using hyper-threading), and bound "widely" when spread between the sockets (avoiding hyper-threading). Fig. 7a shows results using 16 s/w threads, and Fig. 7b shows results for 32 s/w threads. The dom_bc results are for the Domino implementation of BC (note that Domino does not support passive synchronization; we plot the same result for both settings). The bc—triangle_counting results are the Green-Marl benchmarks. The remainder are SPEC OMP.

The single-job behavior of these benchmarks is dependent on the number of threads used, and on the thread placement policy. However, we can make some general ob-

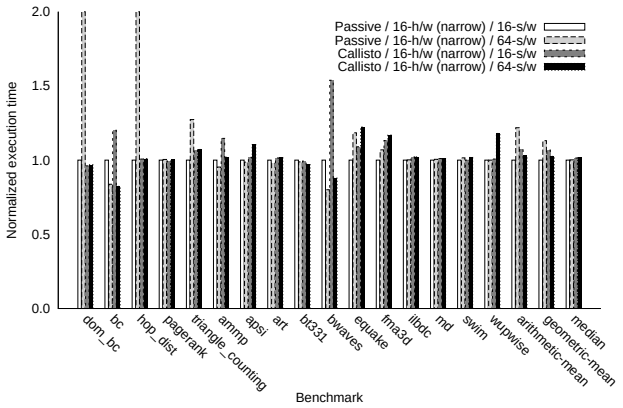


Fig. 9. Running a single job with/without Callisto (truncating the slowest results at 2.0x).

servations: As usual, active synchronization behaves well when hyper-threading is avoided. This trend is illustrated by comparing the active/passive 16-wide results, and the 16-unbound results (the default scheduler tends to distribute threads widely). Passive synchronization tends to be better when using hyper-threading (comparing the active/passive 16-narrow results, and the 32-h/w results). In Fig. 7a, note how the active 16-narrow results are particularly poor for bc and bwaves. These cases exhibit the most load imbalance: more time is spent in synchronization, and so the combination of active synchronization and hyper-threading does the most harm. Note that all of the benchmarks scale to use all 32 h/w contexts. (We omit results for active synchronization with 32 unbound threads; performance is variable and poor, with frequent preemption). From these results, we chose to focus on passive synchronization—there is generally little benefit from active synchronization, particularly when the machine is fully loaded.

Our second set of results varies the number of s/w threads using the original runtime systems (Fig. 8a) and using Callisto (Fig. 8b). Overall, over-subscription does not work well with the original systems. The dom_bc results are poor: the Domino runtime system does not support passive synchronization, and threads are often being preempted while trying to synchronize. In particular, when adding tasks to the deferred work pool, only the thread that added a task is subsequently able to move it to the main work pool. Tasks can get "stuck" if this thread is not scheduled. Over-subscription helps bc and bwaves which exhibited load imbalance. The equake results get worse with extra threads: equake includes a sequential section whose time grows with the thread count. Finally, the performance of hop_dist is much worse with over-subscription (over 4x normalized execution time at 128 threads). This program uses its own spinlock to synchronize updates to nodes in the graph. With over-subscription, OpenMP threads are frequently preempted while holding these spinlocks.

Fig. 8b shows performance with Callisto. As before, we reduce load imbalance in bc and bwaves, and harm equake by adding sequential work. The dom_bc and hop_dist benchmarks are not harmed by over-subscription: the additional runtime system threads are multiplexed over the Callisto workers allocated to the job, and switches between OpenMP threads occur only when the previous runtime system thread blocks or yields. Consequently, runtime system threads are not preempted while holding the unique reference to a piece of work (dom_bc), or spinlocks (hop_dist).

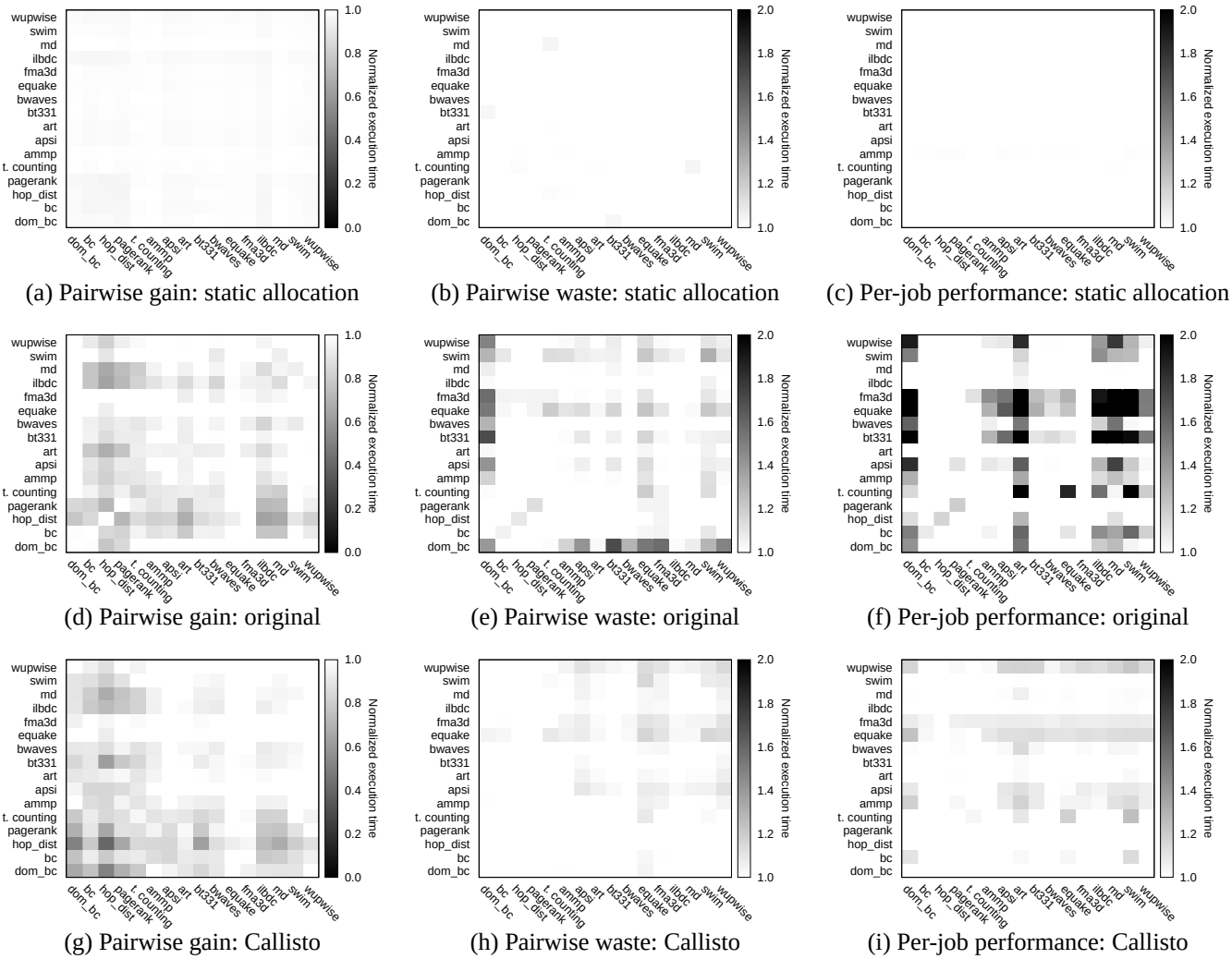
Finally, Fig. 9 provides an overall comparison between the original systems and the Callisto-enabled versions. We show results for 16 h/w contexts, and both with and without over-subscription. The normalized execution time for Callisto with over-subscription ranges from a 22% slow down to a 22% speed up. Overall, there is a slight overhead from using Callisto – the arithmetic mean slowdown is 3.1%, the geometric mean is 2.6%, and the median slowdown is 1.7%.

4.2 Pairs of jobs

We now turn to the performance of pairs of jobs. We compare: (*i*) statically partitioning the machine, giving one socket to each job, (*ii*) running each job with the original runtime systems and 32 unbound s/w threads threads using passive synchronization where available, and (*iii*) running each job with Callisto and over-subscription to 64 s/w threads. Each job is repeated in a loop, continuing until both benchmarks have run at least 5 times. Hence, if a short benchmark runs alongside a longer one, the load of 2 jobs remains until both are complete. We picked the configuration of 32 unbound s/w threads because this gave the best results of the alternatives we tested (specifically, we tested using bound threads but passive synchronization, bound/unbound configurations with each job given exclusive access to one h/w context for its main thread, and unbound configurations with over-subscription).

We plot three heat-maps for each configuration (Fig. 10a–10i). The *pairwise gain* and *pairwise waste* results compare the performance of the pair running together sharing 2 sockets against their performance running one after the other on 1 socket. Note that the results are symmetric because we compare the total time for the pair of jobs. A gray square on the gain heat-map shows that the jobs are faster together (e.g., because their peaks and troughs allow them access to more h/w contexts than when alone on 1 socket). A gray square on the waste heat-map means the jobs are slower together (e.g., inopportune preemption harms one or both jobs). Ideally, the gain heat-map will have gray squares, and the waste heat-map will not.

The *per-job* results show when one job in a pair is being harmed by the other. Unlike the pairwise results, this is not symmetric: some jobs are particularly aggressive, while other jobs are particularly sensitive. For each square, we compare the performance of the job on the y-axis when running as a pair on 2 sockets against the original performance



(a) Pairwise gain: static allocation (b) Pairwise waste: static allocation (c) Per-job performance: static allocation

(d) Pairwise gain: original (e) Pairwise waste: original (f) Per-job performance: original

(g) Pairwise gain: Callisto (h) Pairwise waste: Callisto (i) Per-job performance: Callisto

Fig. 10. Interactions between pairs of benchmarks.

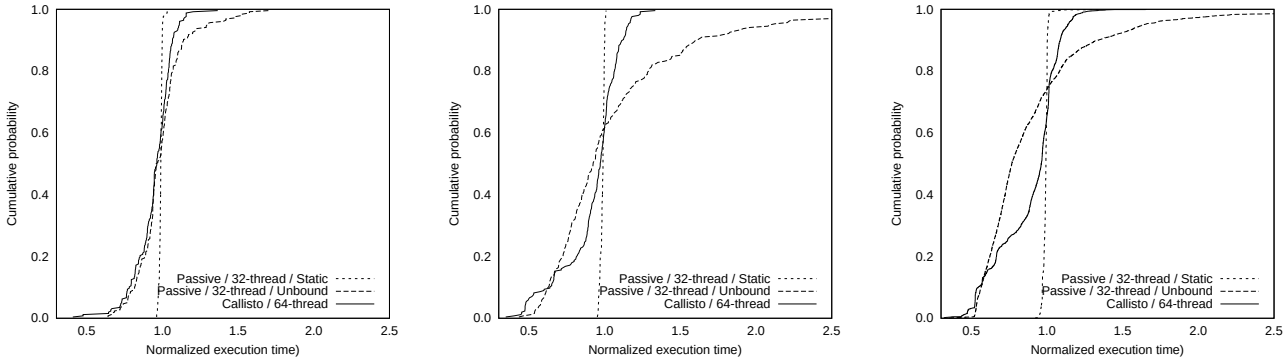
running alone without Callisto on 1 socket. As with our initial results in the introduction, darker squares indicate progressively slower performance. White squares indicate either a speed-up or at least no slow-down. Our plots use the median execution times of each job.

Fig. 10a–10c statically partition the two sockets between the two jobs. The graphs are almost entirely white. There is not much interference, but equally there are no gains.

Fig. 10d–10f show the results without Callisto. The *gain* results show that jobs often benefit from sharing the two sockets. The Green-Marl jobs, in particular, have an IO-bound loading phase during which a concurrent job can use more of the h/w contexts. The *waste* results are poor in pairs with dom_bc—as with over-subscription (Fig. 8) Domino threads are preempted when they hold unique references to deferred tasks. The *per-job* results show that dom_bc, art, ilbdc, md, and swim are all particularly aggressive, with a worst-case slowdown of 3.50x.

The results with Callisto show that there is much less interference, and that when interference does occur it is much less severe. The gain graph (Fig. 10g) shows a similar pattern of gray squares to Fig. 10d: removing interference does not remove opportunities for gains. The waste graph (Fig. 10h) has very few gray squares: the worst is 1.16x for two equake jobs running together, compared with 1.70x for dom_bc and bt331 in Fig. 10e. The per-job results (Fig. 10i) show that Callisto removes most of the interference: the worst execution time is 1.25x (t_counting with swim) compared with 3.50x in Fig. 10f for equake with md.

Finally, note that most of the interference with Callisto occurs in light-gray rows across the heat-map (Fig. 10i), as opposed to columns (Fig. 10f). With Callisto, these results generally match the single-run slow-downs (Fig. 9)—for in-



(a) Pairwise gain and waste (b) Per-job (median run) (c) Per-job (all runs)

Fig. 11. CDFs showing (a) execution time for each pair of jobs, (b) median, (c) all jobs.

stance, equake's performance is degraded because of the sequential work that over-subscription adds.

CDFs of results. Intuitively, the heat maps in Fig. 10 show that Callisto retains the gains when bursty jobs run together, while drastically reducing interference between them. We quantify this by plotting the CDFs for the pairwise and per-job results. Fig. 11 shows three plots, each comparing the different implementations.

Fig. 11a shows the CDF of the pairwise gain and waste results. A normalized execution time below $x = 1.0$ represents a gain, and above $x = 1.0$ it represents waste. The results using Callisto are strictly better than those without (the CDF is always to the left, so more pairs receive lower normalized execution times). The intercept at $y = 1.0$ is also better, showing that the worst-case behavior is improved. (We also checked results using the original runtime systems with over-subscription, and using Callisto without over-subscription: both are worse than the lines we plot.)

Fig. 11b show the CDF of the median per-job execution times (comparing Fig. 10c, Fig. 10f, and Fig. 10i). Again, at $y = 1.0$, Callisto has a drastically reduced tail of long execution times. Above $x = 1.0$, the line for Callisto is to the left of the original: it reduces the number and severity of long execution times. Below $x = 1.0$, Callisto is often to the right of the original. This is expected: without Callisto, aggressive jobs were performing unexpectedly well. Note from Fig. 11a that the pairwise gain results are strictly better with Callisto: the reduction in unfairness is not coming at the cost of overall pairwise performance. The final CDF (Fig. 11c) shows results from *all* job runs, rather than just the median. It shows similar trends to Fig. 11b, confirming that improvements to the median do not occur at the expense of worse outliers.

Impact of Domino on results. Our single Domino workload is particularly sensitive to lock-holder-preemption problems. We therefore tested the sensitivity of our conclusions to the inclusion of Domino. Even without Domino, there are slight reductions in waste in pairs of OpenMP jobs. However, when a pair of these OpenMP jobs run together with passive synchronization, the problem is generally that they receive unfair allocations of CPU time, rather than that they waste resources by spinning. When looking at the unfairness between jobs (Fig. 10f and Fig. 10i), Callisto is effective in removing this unfairness.

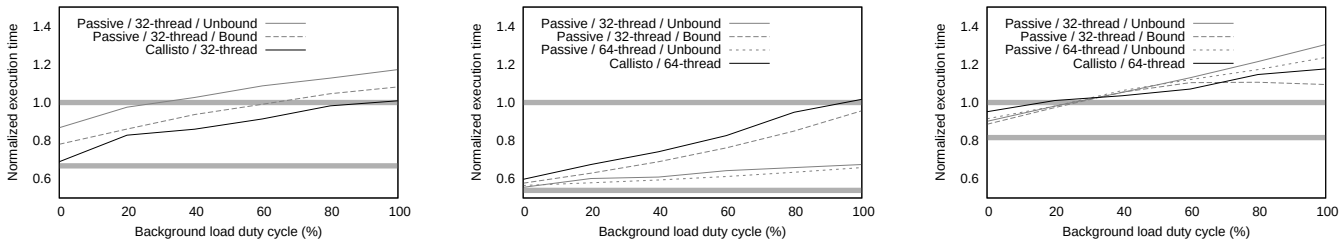
4.3 Synthetic Background Workload

Our final results use a synthetic workload to let us evaluate Callisto under controlled conditions (Fig. 12). The synthetic workload has bursts of CPU demand with a variable duty cycle operating within a 100ms window. We plot results from three representative benchmarks: dom_bc (for the graph analytic algorithms), ilbdc (for most SPEC OMP workloads), and fma3d (for the jobs with an overhead using Callisto).

The thick horizontal lines show the performance when running alone using the original runtime systems with 16 s/w threads (narrow) and 32 s/w threads. For dom_bc and ilbdc the lines for Callisto degrade from the 32-way performance to the 16-way performance as the load increases. For fma3d, the performance of Callisto is offset from the original, reflecting the overhead incurred, but it degrades smoothly. The original runtime systems show interference from the main thread of the synthetic workload (dom_bc), or a great deal of sensitivity to the OpenMP scheduling settings (ilbdc).

Sensitivity to parameters. By default the hysteresis parameters H_{high} and H_{low} are 10ms. Increasing these pushes the Callisto results closer to static partitioning (by allowing a job to retain h/w contexts it is not using). Increasing them to 1s means that only very coarse gains are possible, due almost entirely to IO-bound loading phases. Reducing below 1ms led dom_bc and swim to become more aggressive to other jobs (the other job would more frequently yield a context, and need to wait for it to be returned).

The default low-priority check in interval P_{low} is 1ms and P_{high} is 100ms. Our results do not seem very sensitive to these settings; the check-in test is cheap (looking at a table is shared-memory), and performance seems more dependent on letting jobs retain a h/w context during short gaps in



(a) dom_bc (b) ilbdc (c) fma3d

Fig. 12. Three representative examples when running benchmarks alongside a synthetic job.

their own execution, rather than on letting them regain a h/w context quickly upon unblocking.

We set the runaway timer period to match P_{low}. We felt the timer was important, but it ultimately had little effect on the results; the jobs check in sufficiently often that only occasional long batches of loop iterations are caught.

5. Related Work

Supporting diverse parallel abstractions. Lithe [27, 28] allows multiple runtime systems to coexist in a single application. It introduces a hierarchy of schedulers which split h/w contexts between different parallel frameworks. Callisto operates between jobs, rather than within a single job.

As with Lithe, parallel frameworks such as Manticore [10] and the Glasgow Haskell Compiler [21] support rich sets of abstractions for multiple parallel programming models. Unlike Lithe, the implementations of these abstractions are often tightly integrated with one another [18].

Controlling interference in multiprocessor systems. Zhuravlev *et al.*'s survey [38] describes techniques to reduce contention between h/w contexts in a core, contention between cores sharing a cache, and contention for DRAM channels. Callisto does not currently consider these kinds of contention. Our initial focus has been scheduler interference which has seemed more significant for our workloads.

Linux application containers reduce interference between programs (`https://github.com/google/lmctfy`). They are built over "control groups" to which h/w contexts and memory are allocated. This reduces interference but cannot redistribute resources between fine-grained bursty jobs.

Concurrent with our work, Creech *et al.* proposed the "Scheduling and Allocation with Feedback" system (SCAF) for controlling the number of threads used by concurrent jobs [6]. A daemon periodically updates job-to-core allocations based on estimates of the parallel efficiency of the jobs. SCAF's runtime system is simpler than Callisto's, however it only supports "malleable" OpenMP applications, such as those using dynamically scheduled loops.

Scheduling in multiprogrammed systems. Edler *et al.* proposed application-supplied non-preemption hints [8]. Solaris provides a `schedctl` facility for expressing these hints. We take the opposite approach and identify good preemption points—e.g., between loop iterations.

Tucker provided an early analysis of interference between processes, and proposed a "process control" mechanism to tune the number of runnable threads to the number of h/w contexts [33]. It reacts primarily when jobs start/finish, using a process's internal thread queue to indicate the number of h/w contexts that it can use. Callisto handles burstier workloads which benefit from frequent adaptation, and which use work-share constructs rather than explicit threads.

Majumdar *et al.* [20] and Zahorjan and McCann [37] examine various multiprocessor scheduling policies analytically and via simulation. Their results support the use of dynamic spatial allocations, allowing jobs to vary the number of h/w contexts during their execution. McCann *et al.* confirmed these results in practice in the DYNIX operating system on a 20-processor machine [23].

Marsh *et al.* [22] and Anderson *et al.* [1] designed mechanisms for interfacing user-level thread libraries with the OS kernel. Both allow most work to be done without mode crossings, involving the kernel when blocking in system calls, or when changing the resources allocated to a process. We wished to avoid modifying the OS kernel in our initial prototype, but these mechanisms would let Callisto better support IO operations within jobs.

Recent research operating systems such as Barrelfish [2], fos [35], Akaros [31] and Tessellation [5, 19] have revisited the question of how to schedule parallel jobs.

In Peter *et al.*'s proposed scheduling framework for Barrelfish [29], a job would receive fine-grained gang-scheduled allocation of CPU time across multiple cores. Our experience has been that gang scheduling is not necessary for our workloads, with control over preemption being sufficient to avoid lock-holder preemption problems, and control over machine-wide allocation of cores to jobs being sufficient to avoid unfairness. Gang scheduling may be more important in Barrelfish to support efficient programs based on message passing rather than shared memory.

In a factored operated system (fos) [35], spatial scheduling replaces time multiplexing, on the assumption that the number of cores on a chip will be comparable to the number of active threads. This property is not true in our workloads,

in which individual jobs are able to scale sufficiently well to use the entire machine. Consequently, we move away from a pure spatially-scheduled model.

Akaros [31] and Tessellation [5, 19] enable OS-level coordination across jobs. Both use two-level scheduling of gang-scheduled resource containers. Within containers, they can use Lithe. Resources are divided between containers adaptively based on performance metrics (Tessellation) and resource provisioning (Akaros). This avoids scheduler-related interference. Callisto and Tessellation are complementary: Tessellation makes longer-time-frame external resource trade-offs, which could be followed by short-scale Callisto-style trade-offs for jobs which have allocations on overlapping h/w contexts.

Grand central dispatch (GCD) schedules tasks within an application over thread pools (`http://developer.apple.com/technologies/mac/snowleopard/gcd.html`). The GCD implementation handles sizing the thread pools in response to system load. BWS [7] explores related ideas for runtime systems based on work stealing. The number of threads trying to steal work varies based on their recent success in stealing, and whether or not the process has had threads preempted while executing work.

6. Discussion, Future Work and Conclusions

In this paper we have presented Callisto, and shown how it can drastically reduce the interference between bursty parallel jobs. We have three main directions for future work:

Deployment in practice. Now that we have seen that our overall approach seems effective, we wish to revisit our assumption of cooperation. In some settings cooperation seems effective and reasonable: e.g., within a private cloud, and particularly when a small number of different runtime systems are used. However, in general, it would be prone to several problems—for instance, jobs could corrupt the state that is shared between them, or they could simply not cooperate with requests to change their resource usage.

In the future we want to explore ways to protect the shared state that Callisto uses. This could be done by replacing the current direct access to a shared memory segment with access via a system-call API, or with memory that is shared pair-wise between a single job and a central Callisto management process.

In addition, we wish to introduce external enforcement that jobs "play fair" and relinquish resources when requested. One possibility is to use Callisto to reserve a subset of the cores in a machine: a job can only use these cores if it cooperates. Finally, with this approach, we hope to allow sharing of h/w contexts between jobs running in different virtual machines by treating the reserved h/w contexts as "compute cores" which are managed separately from VMs' normal activity.

Effective resource usage. We want to explore integration with resource management techniques such as those of Bird and Smith [3] to control the division of resources between jobs. Our current policies provide coarse-grained control over the relative number of h/w contexts to provide each job, but they do not attempt to assess how well different jobs are using their resources.

Interference within processors and the memory system. Finally, since we now avoid most scheduler-related interference between jobs, we want to consider the impact of lower level interference of the kind Zhuravlev *et al.* have studied [38]—for instance, finding pairs of s/w threads which would interact well when sharing h/w contexts on the same core, or identifying when it is better to leave a h/w context unused to reduce contention.

Acknowledgments

We would like to thank the reviews and our shepherd Bryan Ford along with Alex Kogan, Dave Dice, John Kubiatowicz, Margo Seltzer, Simon Peter, Sven Stork, and Victor Luchangco for feedback on earlier drafts of this paper.

References

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP '09: Proc. 22nd Symposium on Operating Systems Principles*, pp. 29–44.

[3] S. L. Bird and B. J. Smith. PACORA: Performance aware convex optimization for resource allocation. In *HotPar '11: Proc. 3rd USENIX Conference on Hot Topics in Parallelism*.

[4] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, and A. Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, 21(2):246–254, 1994.

[5] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanović, and J. D. Kubiatowicz. Tessellation: refactoring the OS around explicit resource containers with continuous adaptation. In *DAC '13: Proc. 50th Annual Design Automation Conference*.

[6] T. Creech, A. Kotha, and R. Barua. Efficient multiprogramming for multicores with scaf. In *MICRO '13: Proc 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 334–345.

[7] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. BWS: balanced work stealing for time-sharing multicores. In *EuroSys '12: Proc. 7th ACM European Conference on Computer Systems*, pp. 365–378.

[8] J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. In *Proc. 1988 USENIX Workshop on UNIX and Supercomputers*.

[9] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: scalable nonzero indicators. In *PODC '07: Proc. 26th ACM Symposium on Principles of Distributed Computing*, pp. 13–22.

[10] M. Fluet, M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *ICFP '08: Proc. 13th Conference on Functional Programming*, pp. 241–252.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proc. 1998 Conference on Programming Language Design and Implementation*, pp. 212–223.

[12] T. Harris, Y. Lev, V. Luchangco, V. Marathe, and M. Moir. Constrained data-driven parallelism. In *HotPar '13: Proc 5th Workshop on Hot Topics in Parallelism*.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *NSDI '11: Proc. 8th Conference on Networked Systems Design and Implementation*, pp. 22–22.

[14] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. In *ASPLOS '12: Proc. 17th Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 349–362.

[15] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. A new look at the roles of spinning and blocking. In *DAMON '09: Proc. 5th Workshop on Data Management on New Hardware*, pp. 21–26.

[16] R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling contention management from scheduling. In *ASPLOS '10: Proc. 15th Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 117–128.

[17] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 211–222.

[18] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach. Lightweight concurrency primitives for GHC. In *Haskell '07: Proc. SIGPLAN Workshop on Haskell*, pp. 107–118.

[19] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: space-time partitioning in a manycore client OS. In *HotPar '09: Proc. 1st USENIX Conference on Hot Topics in Parallelism*.

[20] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in multiprogrammed parallel systems. In *SIGMETRICS '88: Proc. Conference on Measurement and Modeling of Computer Systems*, pp. 104–113.

[21] S. Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly, 2013.

[22] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *SOSP '91: Proc. 13th Symposium on Operating Systems Principles*, pp. 110–121.

[23] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.

[24] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[25] J. Nelson, B. Myers, B. Holt, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. Technical Report UW-CSE-14-02-01, University of Washington, 2014.

[26] J. Ousterhout. Scheduling techniques for concurrent systems. In *ICDCS '82: Proc. 3rd Conference on Distributed Computing Systems*, pp. 22–30.

[27] H. Pan, B. Hindman, and K. Asanović. Lithe: enabling efficient composition of parallel libraries. In *HotPar '09: Proc. 1st USENIX Conference on Hot Topics in Parallelism*, p. 11.

[28] H. Pan, B. Hindman, and K. Asanović. Composing parallel software efficiently with Lithe. In *PLDI '10: Proc. 2010 Conference on Programming Language Design and Implementation*, pp. 376–387.

[29] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe. Design principles for end-to-end multicore schedulers. In *HotPar '10: Proc. 2nd Conference on Hot Topics in Parallelism*.

[30] S. L. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In *IARCS '08 Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 383–414.

[31] B. Rhoden, K. Klues, D. Zhu, and E. Brewer. Improving per-node efficiency in the datacenter with new OS abstractions. In *SOCC '11: Proc. 2nd ACM Symposium on Cloud Computing*.

[32] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys '13: Proc. 8th European Conference on Computer Systems*, pp. 351–364.

[33] A. Tucker. *Efficient Scheduling on Shared-Memory Multiprocessors*. PhD thesis, Stanford University, Department of Computer Science, 1993.

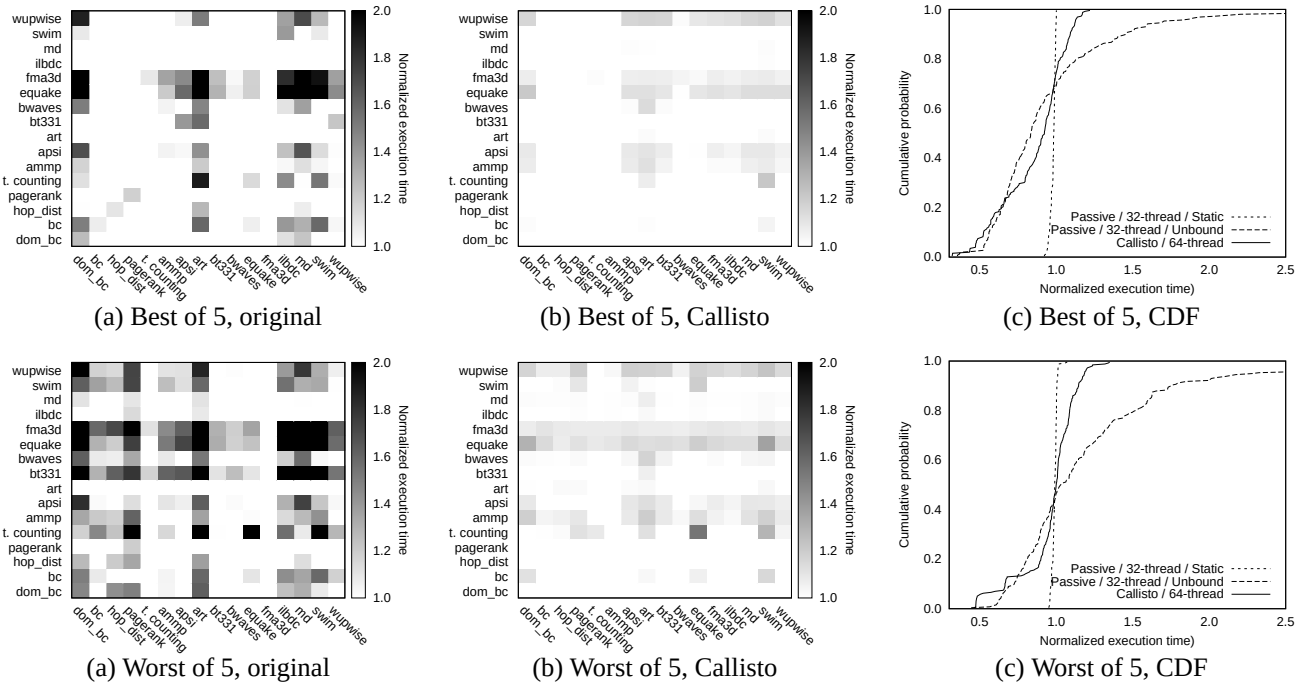
[34] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[35] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[36] J. Zahorjan, E. D. Lazowska, and D. L. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proc. '88 International Seminar on Performance of Distributed and Parallel Systems*, pp. 455–472.

[37] J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *SIGMETRICS '90: Proc. Conference on Measurement and Modeling of Computer Systems*, pp. 214–225.

[38] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1):4:1–4:28, 2012.



(a) Best of 5, original (b) Best of 5, Callisto (c) Best of 5, CDF

(a) Worst of 5, original (b) Worst of 5, Callisto (c) Worst of 5, CDF

Fig. 13. Per-job results at the fastest and slowest runs.

Appendix: Additional Results

This appendix presents additional results, not part of the canonical version of the paper available at `http://dx.doi.org/10.1145/2592798.2592807`. Aside from this addition, the paper is unchanged.

The per-job results in Section 4.2 show the median performance of the jobs over 5 runs. In contrast, Fig. 13 shows results from the same experiments if we focus on the best-of-5 or worst-of-5 results for each pair. Without Callisto, even the best-of-5 results show several cases of 2.5x slow downs. With Callisto the worst-of-5 results remain close to the median, and no worse than 1.53x (t_counting and equake). Hence Callisto is effective in avoiding the most extreme interference between pairs of jobs, not just at reducing the interference in the average case.