MCFlow: a Real-time Multi-core Aware Middleware for Dependent Task Graphs

Huang-Ming Huang, Christopher Gill, Chenyang Lu
Department of Computer Science and Engineering, Washington University
St. Louis, MO, USA
{*hh1, cdgill, lu*}*@cse.wustl.edu*

***Abstract*—Driven by the evolution of modern computer architectures from uni-processor to multi-core platforms, there is an increasing need to provide light-weight, efficient, and predictable support for fine-grained parallel and distributed execution of soft real-time tasks with end-to-end timing constraints, modeled as directed acyclic graphs whose edges capture dependences among their subtasks. At the same time, there is a need to support state of the art programming models such as distributed components, whose ability to encapsulate functionality and allow context-specific optimizations is essential to manage the increasing complexity of modern distributed real-time and embedded systems and systems-of-systems.**

Real-time distributed middleware such as RT-CORBA has not kept pace with these developments, and a new generation of middleware is needed that can map these *dependent subtask graphs* onto distributed hosts with multi-core architectures, efficiently and within a simple, lightweight, and intuitive component programming model. To overcome these limitations, we have designed and implemented MCFlow, a novel distributed real-time component middleware for dependent subtask graphs running on multi-core platforms.

MCFlow provides three new contributions to the state of the art in real-time component middleware: (1) a very lightweight component model that facilitates system integration and deployment through automatic code generation at compile-time from a deployment plan specification; (2) transparent optimization of inter-component communication; and (3) the use of interface polymorphism to separate functional correctness from data copying and other performance constraints so that they can be configured and enforced independently but in a type-safe manner. Empirical evaluations of our approach in comparison to the widely used TAO real-time middleware show that MCFlow performs comparably to TAO when only one core is used and outperforms TAO when multiple cores are involved.

***Keywords*-real-time component middleware; multi-core aware middleware; distributed and parallel real-time subtasks**

I. Introduction

As computers have evolved from uni-processor to multi-core platforms, traditional real-time distributed middleware approaches such as RT-CORBA [1] have not kept pace with that evolution. For example, traditional real-time middleware architectures involve multiple layers of event handling, explicit concurrency management, and synchronization control, which may scale poorly as the number of cores in a host increases, and may impede the middleware's ability to perform at very stringent time scales.

The emergence of multi-core platforms also makes new applications, such as high fidelity real-time testing of civil structures [2], possible through parallel execution of computational subtasks. As was noted in [3], [4], for example, the scale of civil structures such as buildings and bridges often makes it infeasible to test them fully through empirical techniques alone so that sensing and control of physical elements (e.g., a portion of a structure under test and the set of actuators used to impose forces on it [3], [4]) often must be integrated in real-time (e.g., at time scales around 1KHz or faster) with numerical computations for which parallel execution of their subtasks is crucial to meet timing (e.g., to preserve realistic physical dynamics) and scalability constraints. At the same time, component-based development of such systems offers significant advantages in managing the complexity of configuring and integrating different combinations of separately developed functionality for different experimentation and testing scenarios.

A new generation of lightweight real-time component middleware is thus needed that can support and optimize parallel execution of subtasks from multiple real-time tasks, in distributed systems spanning multiple hosts and multiple processor cores within each host. In this paper, we present MCFlow, which is to our knowledge the first light-weight component middleware designed for real-time parallel execution of dependent task graphs atop modern multi-core hosts. Specifically, MCFlow offers the following three novel contributions to the state of the art in real-time component middleware: (1) *a very lightweight component model* that supports safe and flexible configuration and interconnection of components, and facilitates system integration and deployment through automatic code generation at compile-time from a deployment plan specification; (2) *transparent optimization of inter-component communication* so that components can execute in parallel or in series on the same core, on different cores of the same host, or on different hosts, and are inter-connected by as efficient a communication mechanism as is allowed by their relative placements; and (3) *the use of interface polymorphism* to separate functional correctness from data copying and other performance constraints so that components can be configured and constraints enforced independently but in a type-safe manner, according to the specific nuances of the components and system resources that are being integrated.

MCFlow thus provides an efficient component model through which computations can be configured flexibly for execution within a single core, across cores of a common host, or spanning multiple hosts. Application components do not need to be modified in order to cope with the different synchronization and communication mechanisms between cores or hosts when the allocation of subtasks to CPUs changes. MCFlow does this for them transparently, and also enforces a strict separation of performance and functional concerns so that they can be configured independently. MCFlow provides a novel event dispatching architecture that where possible (i.e., when components are executing in the same host) transparently uses lock free algorithms to reduce memory contention, CPU context switching, and priority inversion. To our knowledge, no other real-time component middleware is specifically designed to support directed acyclic real-time task graphs in distributed platforms consisting of multi-core hosts. Our empirical evaluation of MCFlow in comparison to TAO [5], a widely used standards-based middleware for distributed real-time applications, shows that MCFlow performs comparably to TAO when only one core is used, and outperforms TAO when multiple cores are involved, due to the transparent optimizations provided by MCFlow.

This paper is structured as follows. Section II introduces the system model we consider. Section III describes requirements for distributed real-time component middleware on multi-core platforms, which motivate MCFlow's component model and middleware architecture presented in Sections IV and V. Section VI describes experiments that evaluate MCFlow's performance and validate its design and implementation. Section VII surveys other work related to this research, and Section VIII offers concluding remarks.

II. System Model

We consider a distributed real-time environment consisting of hosts with one or more processor cores, connected by a common network, atop which run distributed real-time applications made up of tasks represented as directed acyclic graphs (DAGs) in which the vertexes of each DAG represent subtasks and the edges represent precedence constraints among the subtasks. We say that the subtasks from the same task are *dependent* due to their precedence constraints (e.g., a subtask may depend on data output by other subtasks for its inputs). Subtasks may be allocated freely to different cores on different hosts. Thus, dependences between subtasks may need to be enforced within a single processor core, between cores on the same host, or between hosts. We use the term *team* to denote a set of dependent subtasks allocated on the same host, which belong to the same task.

As Figure 1 illustrates, an *initial subtask* does not depend on any other subtask, an *intermediate subtask* has at least one other subtask on which it depends and at least one other subtask that depends on it, and a *terminal subtask* has no other subtasks that depend on it. Our system model allows tasks to have multiple initial, intermediate, and terminal subtasks, so that arbitrary DAGs are supported.

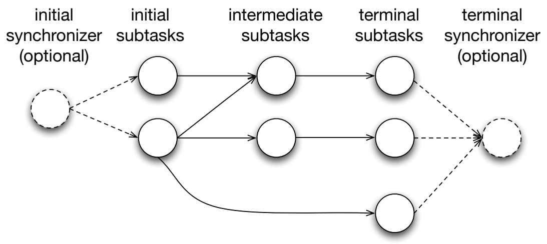
initial synchronizer (optional)
initial subtasks
intermediate subtasks
terminal subtasks
terminal synchronizer (optional)

Figure 1: Task model in MCFlow

However, to meet application synchronization requirements or even to simplify analysis, for many systems it may be useful to consider the common special cases where a task has a single initial subtask, a single terminal subtask, or both. The component encapsulation model and middleware architecture discussed in Sections IV and V make it straightforward to adapt any general task DAG to have a single initial subtask and a single terminal subtask by connecting an optional *initial synchronizer* subtask to the task's initial subtasks, and connecting the task's terminal subtasks to an optional *terminal synchronizer* subtask, as shown in Figure 1. For simplicity, we henceforth assume that each task has a single initial subtask and a single terminal subtask, unless stated otherwise.

Each invocation of a task is called a *job*, and each invocation of a subtask is called a *subjob*. Each job $J_{i,k}$ of task τ_i has a specified release time $t^r_{i,k}$ after which the subjobs of its initial subtasks may begin execution, and a relative deadline $D_{i,k}$ after which processing of the subjobs of all of its terminal subtasks must be completed. Each subjob $J_{i,j,k}$ of each subtask $\tau_{i,j}$ of task τ_i also has an individual release time $t^r_{i,j,k}$ after which it may begin execution.

To allow a direct and efficient mapping onto widely available OS concurrency and scheduling semantics, we assume partitioned fixed priority scheduling of subtasks which execute in threads that are uniquely bound to specific processor cores and do not migrate among them. We also assume that priority assignment and schedulability analysis [6] are performed offline and the results are then encoded in a middleware deployment plan, as described in Section IV-C. These assumptions allow efficient assurance of tasks' end-to-end deadline constraints through enforcement of prioritized execution and periodic release of subjobs, the design and implementation of which are discussed in Sections V-B and V-C respectively.

III. Middleware Requirements

In this section we describe requirements for the design and implementation of real-time component middleware for dependent task graphs on multi-core platforms. These requirements fall into three main categories: *encapsulation*, which guides the component design presented in Section IV, and *real-time capabilities* and *performance optimization* which motivate the middleware architecture presented in Section V.

Encapsulation: Real-time component middleware must provide suitable mechanisms to encapsulate subtasks (hide unneeded details while revealing necessary ones), under a model to which schedulability analysis is readily applied. Real-time component middleware also must separate distinct concerns so that they can be configured independently. To facilitate software reuse, for example, the subtask implementation should be independent from the graph topology. That is, a subtask implementation should be agnostic to whether another subtask is upstream or downstream from it, or to how many immediate upstream or downstream subtasks it has. Subtasks also should be encapsulated to provide location transparency so that the communication between subtasks is independent of how a subtask implementation obtains its input or generates its output. Priorities and other constraints should be specified separately and used by the middleware rather than being entangled within the subtasks' implementations. Type safety also must be enforced between subtasks, i.e., the data sent from an upstream subtask must be acceptable to its downstream subtasks.

Real-time capabilities: Although MCFlow can be configured to support sporadic tasks as well, in this paper we assume a fixed priority periodic task model under which the middleware must provide ways to specify subtask priorities and strictly enforce them when dispatching subjobs, including avoiding any possible priority inversions. In addition, a release guard protocol [6] is used to avoid scheduling anomalies due to subjob release time jitter. The release guard protocol thus enforces an individual release time for each subjob, according to the system model described in Section II.

Performance optimizations: Empirical studies [7], [8] have shown that the costs of thread migration on a multi-core platform can be unpredictable and can introduce meaningful overhead. Therefore, the middleware should avoid thread migration if possible.

Memory sharing among threads may require extensive synchronization and locking control, and may experience unpredictability and additional costs because of cache effects. If memory is shared among cores, timeliness also may be influenced by the cache coherence protocol for which the delay also can be hard to predict. Partitioning the memory used by each middleware thread thus can help reduce the potential costs of cache synchronization, at an additional storage cost that scales proportionally with the product of the number of cores and the number of priorities.

Since resource allocation and deallocation times can be large and unpredictable, the middleware should provide suitable interfaces for reserving required resources in advance. The middleware also should be able to customize communication between subtasks on the same core or on different cores of the same host, rather than always using common inter-process communication (IPC) mechanisms such as sockets or pipes which can incur larger and more variable delays. Furthermore, those optimizations must be provided transparently by the middleware itself so that component implementations remain independent of how they are deployed.

IV. ENCAPSULATION VIA COMPONENTS

To address the encapsulation requirements described in Section III, MCFlow provides a *component model* within which subtask implementations written by application developers are encapsulated. Unlike other component middleware approaches, in which an abundance of features comes at a significant cost in code size and potential run-time overhead and jitter, MCFlow takes a minimalist approach in which each component type is a C++ class that must specify only its inputs, outputs, configuration parameters, runtime execution code, and special interfaces that determine how its input and output types should be initialized. Unlike traditional object oriented middleware frameworks, however, MCFlow does not enforce any inheritance hierarchy on the components but rather uses *interface polymorphism* based on template wrapper classes to encapsulate subtasks so that they can be invoked directly by a dispatcher as is described in Section V-C.

Components in MCFlow are classified into three categories (`source`, `intermediate` and `sink`) depending on whether they generate output and/or consume input data. Every component must provide an associated type called `config_type` and a constructor that accepts a pointer to its `config_type`. This allows developers to control the initial states of their components, such as the maximum size of a matrix or the parameters of a differential equation. The values of these configuration parameters are provided by a deployment plan as is described in Section IV-C.

A. Component Interfaces

MCFlow is designed to support both real-time performance and flexible component-based design. As described in Section III, dynamic memory allocation may introduce high cost and jitter, and yet to forbid the use of dynamic memory at all could seriously impact the flexibility of component design. One standard way to address this issue is to estimate an upper bound on the size of memory required overall, and to preallocate enough memory at initialization time.

This rationale gives rise to the design of MCFlow's component interface: the separation of input and output initialization from the construction of the component. An application can utilize the `config_type` to set the memory size at configuration time. The input and output initialization interfaces allow MCFlow to provide appropriately sized (e.g., as in [9]) input and output *ring buffers* that support the optimizations described in Section V-B. The separation of input and output initialization provides greater freedom for the framework to optimize the input and output buffers without complicating the component interface itself.

Notice that whether to use a maximum memory reservation strategy at component initialization time or to use dynamic memory allocation is left to the discretion of the developer. For systems with more stringent timing constraints advance reservation may be appropriate, while for applications with looser constraints the flexibility offered by dynamic memory allocation may be the dominant concern. MCFlow provides a convenient interface so that a developer can choose which memory allocation strategy to use based on the specific requirements of their system.

B. *Interface Type Safety*

MCFlow also enforces compatibility of both data types and memory management for input and output between subtasks through its component interfaces. For example, the connection between two components is only valid if the upstream component's output type for that connection is assignable to the downstream component's input type and downstream handling of memory buffers will be appropriate. To overcome a potential limitation with this approach on the reusability of components, MCFlow also allows adapters for component connections to be specified. An adapter is a C or C++ function that takes the output of an upstream component and coverts it into the input of a downstream component. MCFlow's connection *ports* are essentially data members of the user defined types named `input_type` or `output_type` within each component class. Instead of copying the entire output from an upstream component to a downstream component, a port allows the application developer to selectively connect part of an upstream component's output to all or part of a downstream component's input as long as the connection is type safe.

Since we allow components that are run in the same thread to share memory via their input and output interfaces, the lifetime of the validity of the shared memory becomes a potential issue. In MCFlow, each job execution is implicitly associated with a sequence number that is used to index the corresponding ring buffer for input and output queues. As long as the queue size is greater than the maximum pipeline level of any task, the output data of the first subtask won't be overwritten until the last subtask of the task has finished its work. Therefore the memory passed from an upstream subtask will always be valid until it finishes its current job execution. However, it is not safe to save a pointer to the memory and use it for the next occurrence of the job. In that case, the component should always copy the memory contents into its local state variables.

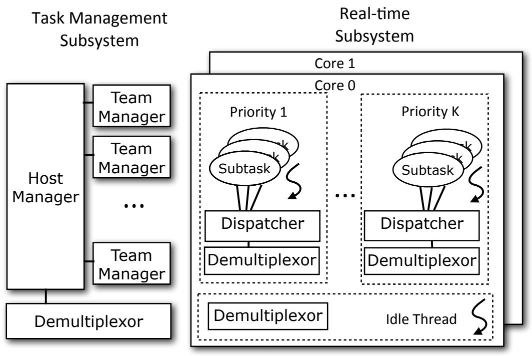

Figure 2: MCFlow Host Architecture (the squiggly arrows represent threads)

C. *Deployment Plan and Code Generation*

To make connections between subtasks even more transparent, MCFlow provides a *deployment* tool which reads the specification of a deployment plan and generates appropriate C++ source files and Makefiles according to its contents. A deployment plan's specification includes: (1) the hosts in the execution environment and their network addresses; (2) all tasks and the subtasks they contain; (3) for each subtask the type of component used, the values for each field in the `config_type`, on which host and core the subtask should be executed, and the priority of the subtask; and (4) the connections between the subtasks.

V. Middleware Architecture

MCFlow enforces a crucial separation of concerns between its task management and dispatching subsystems, as Figure 2 illustrates. The task management subsystem creates, initializes and terminates the subtasks on each host. If a subtask throws an unrecoverable exception, the task management subsystem releases all resources previously acquired by that subtask's team. The dispatching subsystem is designed specifically to enforce real-time requirements and apply performance optimizations discussed in Section III. In particular, all threads in the dispatching subsystem, and the memory resources they use, are strictly and unchangeably pinned to a specific priority and processor core. In both the task management and dispatching subsystems, events (e.g., from network I/O) are demultiplexed onto threads using the Linux `epoll` APIs.

A. *Task Management Subsystem*

The creation and destruction of subtasks in a host is done by the task management subsystem. The host where a task originates creates the subtasks assigned to it and issues initialization requests to other hosts in the system. Upon acknowledgement from the downstream hosts, it activates its

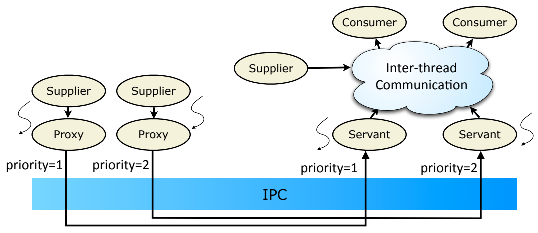

Figure 3: Real-time Communication via Priority Lanes

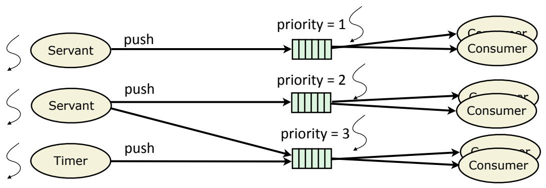

Figure 4: TAO Real-time Event Service ITC Mechanism

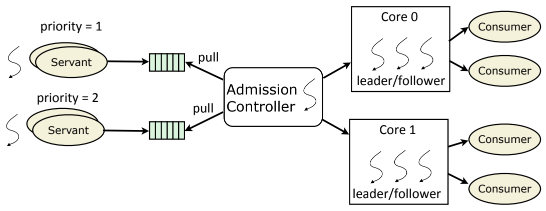

Figure 5: MC-ORB ITC Mechanism

local task dispatching subsystem to start real-time execution of the task.

A termination protocol for each task is executed at the task's real-time priority, to ensure that all the subjobs have stopped executing and their subtasks can be safely deallocated. Termination may be initiated either by an upstream data source or by run-time exceptions from downstream subtasks. During team termination, the team manager first sends a termination request to all the downstream subtasks, to be executed at their real-time priorities. Upon receiving the termination request, a subtask will stop accepting any new inputs and will pass the request to its successors. In addition, it sends an asynchronous notification[1] to the team manager to indicate that the subtask has stopped. Once the team manager receives all notifications from all subtasks in the team, it can deallocate resources reserved for the team.

B. Dispatching Subsystem

The dispatching subsystem prioritizes execution of subtasks using priority lanes as is shown in Figure 3. Each priority lane is a collection of threads which are allowed to share resources without contributing to priority inversion [10]. Instead of using a dedicated thread with a fixed priority to receive IPC messages, it allocates multiple threads to receive and handle messages at different priorities. Each supplier or consumer in the figure represents an upstream or downstream subtask, and each proxy or servant actually sends or receives data through an IPC channel. Priority lanes are widely used to avoid inter-process priority inversion, though different inter-thread communication (ITC) mechanisms may be used to deliver messages from servants to consumers.

For example, TAO's Real-time Event Service [11] uses the ITC mechanism shown in Figure 4 to support event multicast, filtering, and correlation, through queues for consumers at different priorities within a single processor.

To support allocation of threads onto multiple processors, MC-ORB [8] uses a half-sync/half-async concurrency architecture [12] for receiving network requests, in which a dedicated thread decides in which core and at which priority each request should be handled, and then pushes each request to a designated thread pool as is shown in Figure 5. This supports thread migration and allows adaptive run-time load balancing of subtasks, but suffers two additional context switches for each subtask (in contrast to our system model in which requests are delivered directly to subtasks), which may impact timing guarantees especially with large numbers of subtasks.

MCFlow instead relies on static task allocation at configuration time so that it can optimize inter-thread communication. For each subtask $T_{i,j}$, each of its immediate predecessor subtasks is given a distinct network address and identifier for the host and core where $T_{i,j}$ should be run. The network request is then delivered directly to the appropriate core, rather than going through an intermediate thread to dispatch the request. This design choice allows efficient dispatching of network events without the extra context switches seen in MC-ORB.

Neither TAO's Real-time Event Service nor MC-ORB optimizes inter-core communication, and therefore the data exchanged between subtasks must be marshaled and demarshaled even if the suppliers and consumers are on the same host. This can be very inefficient if a large amount of data needs to be exchange among dependent subtasks, which is a common case for computations like the numeric simulations mentioned in Section I. Other alternatives like direct function calls passing native C++ pointers or reference counted smart pointers are possible within a single process, but they may reduce parallelism and incur variable costs for dynamic memory allocation and deallocation.

To alleviate the need for either data marshaling and demarshaling or memory allocation and deallocation, MCFlow uses the novel ITC mechanism shown in Figure 6. Unlike TAO's Real-time Event Service or MC-ORB where consumer side queues are shared, in MCFlow each consumer has its own type-specific input queue and each supplier has its own type-specific output queue. Since both input

[1]We use an `eventfd` that is provided by the current Linux kernel for lightweight event notification; it is also possible to implement asynchronous notification using a pipe in an older kernel, but with a higher cost.

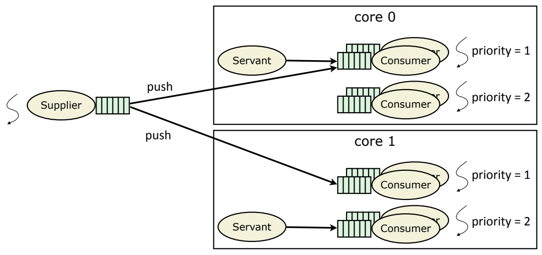

Figure 6: MCFlow ITC Mechanism

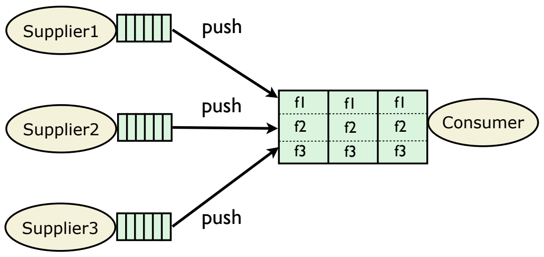

Figure 7: Data Merge in MCFlow

queues and output queues are type-specific, data can be copied directly between them in a type-safe manner without marshaling and demarshaling.

MCFlow uses a simple lock free ring buffer to avoid mutex synchronization. If a consumer has multiple suppliers, the consumer will be dispatched if and only if all its suppliers have copied data into its input queue. Each entry in the consumer queue may have multiple data fields, where each field is used for a specific supplier as shown in Figure 7.

To synchronize data merging from multiple suppliers, each job has a sequence number that indexes the input and output queues. The supplier queues are also implemented as ring buffers, and use this index to retain the data for as long as the consumers need it. A sufficient size for the supplier queue can be calculated automatically [9] by the deployment tools described in Section IV-C, from the supplier subtask's relative deadline, its period, and the depth of the task graph in that host. The memory passed from an upstream subtask moves from queue to queue but will always be valid until the current job finishes execution. MCFlow also ensures that no two elements in a supplier or consumer ring buffer share the same cache line, by padding each element in the buffer so that its size is a multiple of the hardware cache line size, which thus avoids the *false sharing problem* [13] for access to the buffer, to reduce latency of inter-component communication within a host.

C. Subtask Release Mechanism

In MCFlow a dispatcher coordinates all subtasks allocated to a specific core at a particular priority. As Figure 2 illustrates, each dispatcher has an associated demultiplexer for passing subtask invocation requests to it (according to the *reactor* architectural pattern [12] in which asynchronously arriving events, e.g., for network I/O, are demultiplexed by invoking specific event handlers that have registered to receive those events). In MCFlow subtasks are dispatched in this manner, and a demultiplexer can be configured either with a single thread (in which case the different subtasks are dispatched sequentially in FIFO order) or using the leader/followers pattern [12] with multiple concurrent threads waiting for events. The leader/followers configuration can be especially useful when subtasks can block in certain system calls.

Each dispatcher also manages a FIFO subtask queue and a timer queue to control when each subtask can be executed. When a subtask finishes its execution, it copies its outputs to the input queues of its immediate downstream subtasks, as described in Section V-B, and then inserts those subtasks into the subtask queues of their corresponding dispatchers. After that, it sends asynchronous notifications to their designated demultiplexer. Once a demultiplexer thread picks up the notification, it processes the notification in the following steps: (1) the thread removes a subtask from the subtask queue; (2) the thread checks whether that subtask is still being processed (by reading an atomic *in-processing* flag); this step is required when the leader/followers pattern is applied; in this case, multiple threads exist for the same core/priority and two subjobs may be executing concurrently if one thread is still processing a subjob when another notification is sent and is picked up by another thread; (3) if the subtask is not being processed, the thread then checks whether the subtask is periodic and whether the release time for the subtask has expired; (4) if the release time has expired, the *in-processing* flag is set and the subtask is executed in the thread; otherwise the subtask is inserted into the timer queue, to be executed when the timer expires; (5) after the subtask finishes executing, it checks whether there are more inputs to be consumed and keeps executing until no inputs are available; (6) the *in-processing* flag is cleared before the the thread waits again for events.

Step 3 is required to enforce release-guard semantics [6] across distributed or multi-core systems so that *intervals between release times of jobs in any subtask are never less than the period of the subtask* [14]. Besides waiting for the period boundary before the next subjob of a subtask can be executed, the release-guard protocol also allows a subjob to be executed earlier than the periodic boundary when the CPU becomes idle. To implement this feature, another *idle thread* with the lowest real-time priority for each CPU waits on a prioritized demultiplexer. Whenever the timer queue size changes, the dispatcher sends a notification to the idle thread with the new size of the queue. The idle thread can only receive those notifications when there are no other real-time subjobs executing in the CPU. Once the idle thread receives the notification, it will then send an idle notification to the highest priority dispatcher that has a nonzero timer queue size. That dispatcher will then dispatch the subtask with the earliest expiration in its timer queue. The dispatching scenario is similar for network triggered subtasks. In this case, the subtask is notified directly upon readability of the socket instead of upon the receipt of an asynchronous notification.

VI. Performance Evaluation

In this section, we evaluate the performance of MCFlow for dispatching real-time subtasks in parallel. We examine the end-to-end latency for a single flow application and the deadline miss ratios for a multi-flow application.

Of the alternative middleware architectures discussed in Section V-B, we compare MCFlow to the TAO [5] RT-CORBA object request broker, to judge how much of an improvement MCFlow offers compared to a state of the art real-time middleware for distributed real-time systems. We chose TAO as a baseline because it is widely used and is reasonably representative of other real-time ORBs such as nORB [15] and MC-ORB[2] [8].

In Section VI-A, we evaluate how well MCFlow can reduce the latency of a basic distributed real-time task with dependent parallel subtasks. Section VI-B describes experiments designed to examine MCFlow's ability to parallelize subtasks across different number of cores. The multi-flow experiments in Section VI-C evaluate how well MCFlow enforces priorities among different tasks.

The experiments described in this section were performed on two 6-core Intel core i7 980 3.3GHZ hosts with hyper-threading enabled, connected via TCP/IP sockets over 100BASE-T switched Ethernet. Both machines ran Ubuntu Linux 10.04 with the 2.6.33-29-realtime Kernel (with the PREEMPT_RT Patch [16]). In this section, we use *CPUs* to refer to the number of logical processors recognized by the operating system, rather than the number of physical cores.

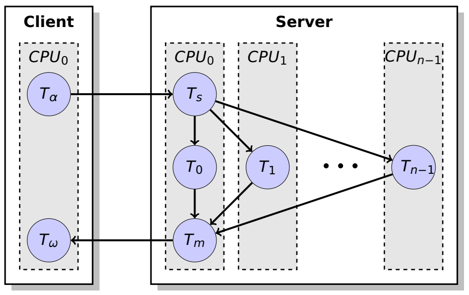

Figure 8: Single Task Experiment Setup

We measure the latency of a task spanning two hosts, as is shown in Figure 8. The server receives data from a client and splits the subtask computations onto a number of CPUs, merges their output into a combined result, and sends it back to the client. To evaluate how computation splitting itself affects performance, we vary the number of subtasks and the number of CPUs used together. We use T_α and T_ω to represent the data source and sink subtasks on the client; T_s and T_m to represent the data splitting and merging subtasks on the server; and T_i (where $i = 0$ to $n-1$) for the n parallel subtasks on the server. The data transmitted from T_s to each T_i, and from each T_i to T_m are 64 bytes long. The data transmitted from T_α to T_s and from T_m to T_ω are $64n$ bytes long. No extra computation is done in T_α and T_ω. The computation times for parallel subtasks T_0 through T_{n-1} (and also T_s and T_m) are all 5 μs.

A. Latency Comparison

We compare the latency of these applications using MCFlow (using release guards, which may increase average but not worst case latency) and TAO. The MCFlow version has two configurations: the first uses all of the optimizations described in Section V, while the second configuration (denoted MCSock) uses only sockets for inter-component communication. These configurations are compared to examine how much the inter-core communication optimizations can improve system performance.

The TAO version also consists of two different configurations. The first uses one ORB per CPU (denoted as TAO-MORB), with each ORB allocated only one thread. Each thread is pinned to a particular CPU to avoid migration. All subtasks are assigned to their corresponding ORBs. The collocation strategy [17] used for this configuration is "per-ORB" which means the requests are optimized to use direct function calls when the caller and callee are registered with the same ORB. The second configuration uses the leader/followers pattern with only one ORB per application and n threads (denoted as TAO-SORB). In this configuration, a subtask can't be run on a fixed CPU. No collocation optimization is used for this configuration; if it did, all CORBA invocations would become function calls and thus the entire server could only run in one thread.

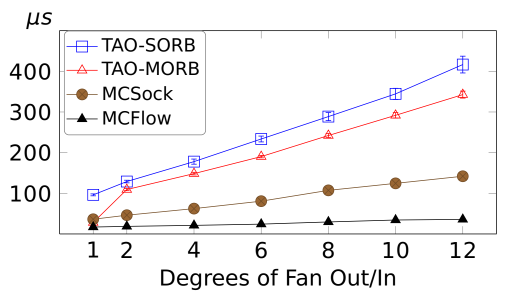

Figure 9: Average Server Response Time Latency

Figure 9 shows the average time from when T_s receives a request to when T_m sends a reply, for different numbers of

[2]While MC-ORB supports core-specific assignment of threads, it also may migrate threads dynamically to balance utilization at run-time, which would violate our system model assumptions described in Section II.

parallel subtasks to which requests are fanned out by T_s (and from which the results are fanned in to T_m). Figure 10 shows the average time from when T_s finishes its own computation to when T_m receives the last message from any $T_i, \forall i = 0, \cdots, n-1$. The error bars in each of these figures show the (small) standard deviation over 10,000 values.

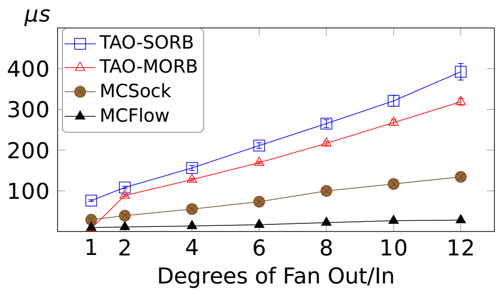

Figure 10: Average Parallel Release to Termination Latency

Figures 9 and 10 show little difference in performance for MCFlow and TAO-MORB when there is only one CPU. However, the latency for each point along the TAO-MORB curve grows far faster than its MCFlow counterpart as the number of cores onto which tasks are split increases. Both MCSock and TAO-MORB use sockets as their only mechanism for inter-core communication, and thus the gap between MCSock and TAO-MORB stems from factors other than inter-core communication. This difference is due at least in part to the smaller memory and CPU footprint of MCFlow compared to TAO since as we note in Section V-B, MCFlow avoids marshaling and demarshaling and other unnecessary features within a single host. In our testing, the sizes of the client and server versions of MCFlow were 700K and 718K bytes; the sizes for TAO were 1793K and 2072K bytes. In addition, the TAO version requires ancillary dynamic link libraries to run while MCFlow requires nothing but the standard libstdc++.

Notice that the single ORB version of TAO always performs worse than the per CPU ORB version: because an ORB maintains resources that need to be synchronized among threads, using only a single ORB may incur significant synchronization overhead. In contrast, the per CPU ORB version duplicates resources to each CPU and thus avoids such resource contention, much in the way MCFlow duplicates resources as we noted in Section V.

Figure 11 shows the average time measured in the client from when T_α sends a request until when T_ω receives a reply. The results shown in Figure 11 demonstrate that end-to-end latency differences between MCFlow and TAO are small with between one to four CPUs. This is largely because the round-trip network communication cost dominates the end-to-end latency. However, as the number of parallel subtasks increases MCFlow's ability to parallelize real-time subtasks efficiently across cores becomes the dominant factor.

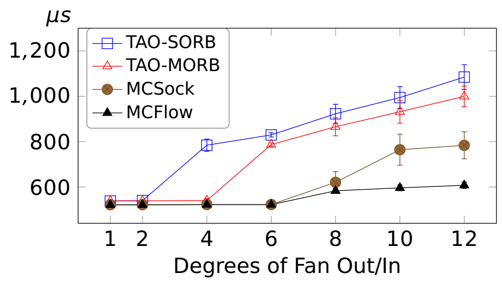

Figure 11: Round Trip Latency Comparison

B. Speedup Comparison

In the previous comparisons, we showed how the communication cost increases when the same amount of data is being transmitted on each connection, so that the total data size increases linearly with the degree of parallelism. The experiment in this subsection uses a fixed workload which does not vary with the number of cores used, and equally splits the work among cores to evaluate how workload spreading can reduce the end-to-end latency.

The experiment setup is similar to that of the previous experiment; however, T_s and T_m are only used to separate and merge data to and from the cores, and no extra workload is generated in those two subtasks. A workload of roughly 1000 μs is divided equally among n cores and processed by parallel subtasks T_0 to T_{n-1}. The data generated by T_α is 480 bytes long and is equally split among T_0 to $T_n - 1$.

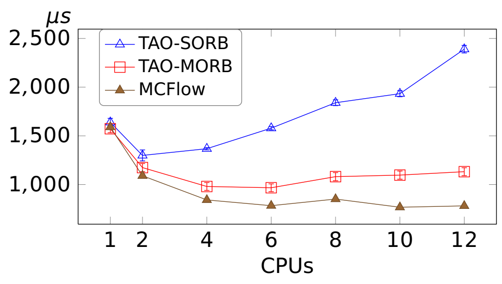

Figure 12: Latency Comparison Server Result

Figure 12 shows that with up to 6 CPUS, the response time for MCFlow and for TAO with one ORB per CPU decreases when more cores are used, with MCFlow again providing lower response times. However, there is a slight increase in response times with more than 6 cores. This is due to the fact that our test machine is actually a 6 core machine with hyper-threading enabled. The response time for the TAO Single ORB configuration increases after 4 cores, again due to the higher synchronization overhead mentioned in Section V.

C. Real-time Performance

We also designed the following experiment to examine how well MCFlow can preserve the priority constraints of an application. In this experiment, we created tasks with three different priorities: high, medium and low. All the subtasks of the high priority task have higher priority than those of the other two tasks; similarly, all subtasks of the low priority task have lower priority than those of the other two.

Similar to the previous experiment, each task spans two hosts and one host is used for a client which only sends periodic requests to server. The server again splits the workload onto multiple CPUs, merges the result and sends it back to the client. In our experiment, all the client subtasks are on the same machine and all server subtasks are on the other. The topology of each task is similar to Figure 8; however, different CPU assignments and workloads are used.

Table I: (CPU, Workload in μs) for each Task's Subtasks

	High	Med	Low
T_s	(0, 900)	(1, 900)	(2, 900)
T_m	(0, 900)	(1, 1800)	(2, 900)
T_0	(0, 1800)	(0, 0)	(0, 1800)
T_1	(1, 1800)	(1, 1800)	(1, 900)
T_2	(2, 1800)	(2, 1800)	(2, 4500)
T_3	(3, 1800)	(3, 1800)	(3, 3600)

Table I shows the CPU assignment and the workload in μs for each subtask of the high, medium, and low priority tasks. The frequencies of the high and medium priority tasks are fixed at 200Hz and 100Hz respectively. We vary the frequency of the low priority task and observe its effect on the rest of system. We assume the deadline of each task is equal to its period. The real-time performance of the tasks in this experiment is summarized in Table II. When the rate of the low priority task is below 70 Hz, there are no deadline misses. With an increase in the low priority task's rate, it begins to miss deadlines but the other two tasks do not. Similarly, the response times of the high and medium priority tasks, shown in Table III, remain stable even as the low priority task's rate increases.

Table II: Tasks Deadline Miss Ratios

	High	Med	Low
50 Hz	0	0	0
60 Hz	0	0	0
70 Hz	0	0	0.06
80 Hz	0	0	0.75
90 Hz	0	0	1.0

Table III: Average Response Times in μs

	High	Med	Low
50 Hz	3918	8127	14918
60 Hz	3929	8065	12615
70 Hz	3926	8063	12881
80 Hz	3931	8129	13574
90 Hz	3928	8045	18919

VII. Related Work

The middleware approach described in [4] is based on the preliminary work from [3] to target real-time hybrid testing of civil structures. It was based on dependent task graphs, but was designed for uniprocessor systems and lacks optimizations for multi-core platforms or the ability to reconfigure graphs of dependent subtasks flexibly and transparently. Achieving those capabilities across multi-core platforms is a key motivation for developing MCFlow and is one of the main contributions of this work.

The OMG Real-Time CORBA specification [1] supports distributed software component development [18] and provides real-time policies and mechanisms including standard interfaces to specify resource requirements and configure object request broker (ORB) end-system resources. TAO [5], a widely used Real-time CORBA ORB, was developed for single processor systems and does not support fine-grain parallelization of subtasks or inter-core communication optimizations on multi-core platforms. TAO's Real-time Event Service [11] supports decoupled communication between objects, which makes it more convenient to implement distributed tasks. However, the event service requires a centralized event dispatcher which may lead to high synchronization overhead and thus may become a bottleneck on multi-core platforms. The MC-ORB [8] middleware was specifically developed for multi-core platforms. However, it is also based on the CORBA remote method invocation paradigm and is designed for task sets without dependences.

Real-time parallel task scheduling has received significant attention recently [19], [20]. Parallel programming languages, extensions, and libraries, such as Cilk [21], OpenMP [22] and Intel Thread Building Blocks [23] provide different forms of support for parallel programs. Charm++ [24] recently has been extended to support many-core architectures [25]. Dataflow programming languages and frameworks such as SystemC [26], StreamIt [27], and FastFlow [28] also have been developed. While each of these approaches has advanced the state of the art in executing parallel and/or real-time task graphs, none of them provides a flexible component programming model through which those tasks may be configured, deployed, and interconnected, which is also a key contribution of this work.

VIII. Conclusions

In this paper we have described the design and implementation of MCFlow, a novel real-time component middleware designed specifically to support subtask parallelization for distributed real-time applications on multi-core platforms. MCFlow provides a simple but flexible component-based development model in which an application developer does not need to program networking or data synchronization semantics directly, but rather uses a deployment tool to specify how components are connected and to give their real-time constraints. MCFlow can optimize communication between components based on their location and connection topology and whether each connection is intra-core, inter-core or across a network. Results of the experiments presented in Section VI show that MCFlow performs comparably to TAO when only one core is used, and outperforms TAO when multiple cores are involved. MCFlow also enforces prioritization constraints effectively so that higher priority tasks are not affected by lower priority ones.

References

[1] Object Management Group, "Real Time-CORBA Specification, Version 1.2," Nov. 2003.

[2] P. B. Shing, Z. Wei, R. Y. Jung, and E. Stauffer, "NEES fast hybrid test system at the University of Colorado," in *13th World Conference on Earthquake Engineering*, Vancouver, Canada, 2004.

[3] T. Tidwell, X. Gao, H.-M. Huang, C. Lu, S. Dyke, and C. Gill, "Towards configurable real-time hybrid structural testing: a cyber-physical system approach," in *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, IEEE. Tokyo, Japan: IEEE, Mar. 2009, pp. 37–44.

[4] H.-M. Huang, T. Tidwell, C. Gill, C. Lu, X. Gao, and S. Dyke, "Cyber-physical systems for real-time hybrid structural testing," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems - ICCPS '10*, no. 2. New York, New York, USA: ACM Press, 2010, p. 69.

[5] Institute for Software Integrated Systems, "The ACE ORB (TAO)," www.dre.vanderbilt.edu/TAO/.

[6] J. Sun, "Fixed-priority end-to-end scheduling in distributed real-time systems," Ph.D. dissertation, University of Illinois, Urbana-Champaign, USA, 1997.

[7] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers," in *Proc. Real-Time Systems Symposium*. IEEE, Nov. 2010, pp. 14–24.

[8] Y. Zhang, C. Gill, and C. Lu, "Real-time performance and middleware for multiprocessor and multicore Linux platforms," in *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, no. 314. IEEE, Aug. 2009, pp. 437–446.

[9] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

[10] I. Pyarali, D. Schmidt, and R. Cytron, "Techniques for enhancing real-time CORBA quality of service," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1070–1085, Jul. 2003.

[11] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time CORBA event service," *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 184–200, Oct. 1997.

[12] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[13] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance," in *Sedms'93: USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems*. San Diego, California: USENIX Association, 1993, pp. 53–71.

[14] J. W. S. W. Liu, *Real-time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[15] V. Subramonian, G. Xing, C. Gill, C. Lu, and R. Cytron, "Middleware specialization for memory-constrained networked embedded systems," in *Proc. Real-Time and Embedded Technology and Applications Symposium*, 2004.

[16] S. Rostedt and D. V. Hart, "Internals of the RT Patch," in *Proceedings of the Linux symposium*, 2007.

[17] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. S. Gokhale, "Applying optimization principle patterns to design real-time ORBs," in *Proceedings of the 5th conference on USENIX Conference on Object-Oriented Technologies & Systems*. USENIX, 1999, pp. 145–160.

[18] *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed., Object Management Group, May 2003.

[19] C. Liu and J. H. Anderson, "Supporting soft real-time dag-based systems on multiprocessors with no utilization loss," in *RTSS*. IEEE, 2010.

[20] I. Lupu and J. Goossens, "Scheduling of hard real-time multi-thread periodic tasks," in *RTNS*, 2011.

[21] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk," *ACM SIGPLAN Notices*, vol. 30, no. 8, pp. 207–216, Aug. 1995.

[22] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[23] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007.

[24] UIUC Parallel Programming Laboratory, "Parallel Languages/Paradigms: Charm++ - Parallel Objects," charm.cs.uiuc.edu/research/charm/.

[25] ——, "Using Charm++ to test future many-core architectures," charm.cs.uiuc.edu/news.html?highlight=true.

[26] T. Grotker, *System Design with SystemC*. Norwell, MA, USA: Kluwer Academic Publishers, 2002.

[27] S. Amarasinghe, M. l. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies, "Language and compiler design for streaming applications," *International Journal of Parallel Programming*, vol. 33, no. 2-3, pp. 261–278, Jun. 2005.

[28] M. Aldinucci, S. Ruggieri, and M. Torquati, "Porting decision tree algorithms to multicore using FastFlow," in *Proceedings of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*, ser. LNCS, J. L. Balcázar, F. Bonchi, A. Gionis, and M. Sebag, Eds., vol. 6321. Barcelona, Spain: Springer, Sep. 2010, pp. 7–23.