Report Number: WUCSE-2010-57

2010

The Design and Implementation of MCFlow: a Real-time Multi-core Aware Middleware for Dependent Task Graphs

Huang-Ming Huang, Christopher Gill, and Chenyang Lu

Modern computer architectures have evolved from uni-processor platforms to multi-processor and multi-core plat- forms, but traditional real-time distributed middleware such as RT-CORBA has not kept pace with that evolution. To address those issues, this paper describes the design and implementation of MCFlow, a new real-time distributed middleware for dependent task graphs running on multi-core platforms. MCFlow provides the following contributions to the state of the art in real-time middleware: (1) it provides an efficient C++ based component model through which computations can be configured flexibly for execution within a single core, across cores of a common host, or spanning... **Read complete abstract on page 2.**

The Design and Implementation of MCFlow: a Real-time Multi-core Aware Middleware for Dependent Task Graphs

Huang-Ming Huang, Christopher Gill, and Chenyang Lu

Complete Abstract:

Modern computer architectures have evolved from uni-processor platforms to multi-processor and multi-core plat- forms, but traditional real-time distributed middleware such as RT-CORBA has not kept pace with that evolution. To address those issues, this paper describes the design and implementation of MCFlow, a new real-time distributed middleware for dependent task graphs running on multi-core platforms. MCFlow provides the following contributions to the state of the art in real-time middleware: (1) it provides an efficient C++ based 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; (2) it allows optimizations for inter-component communication to avoid data copying without sacrificing the parallel executability of data dependent tasks; (3) it strictly separates timing and functional concerns of an application so that they can evolve and can be configured independently; and (4) it provides a novel event dispatching architecture that uses lock free algorithms to avoid mutex locking and reduce memory contention, CPU context switching, and priority inversion. We also present an empirical evaluation that demonstrates the efficacy of our approach.

2010-57

The Design and Implementation of MCFlow: a Real-time Multi-core Aware Middleware for Dependent Task Graphs

Authors: Huang-Ming Huang, Christopher Gill, Chengyang Lu

Corresponding Author: {hh1, cdgill, lu}@cse.wustl.edu

Abstract: Modern computer architectures have evolved from uni-processor platforms to multi-processor and multi-core plat- forms, but traditional real-time distributed middleware such as RT-CORBA has not kept pace with that evolution.
To address those issues, this paper describes the design and implementation of MCFlow, a new real-time distributed middleware for dependent task graphs running on multi-core platforms. MCFlow provides the following contributions to the state of the art in real-time middleware: (1) it provides an efficient C++ based 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; (2) it allows optimizations for inter-component communication to avoid data copying without sacrificing the parallel executability of data dependent tasks; (3) it strictly separates timing and functional concerns of an application so that they can evolve and can be configured independently; and (4) it provides a novel event dispatching architecture that uses lock free algorithms to avoid mutex locking and reduce memory contention, CPU context switching, and priority inversion. We also present an empirical evaluation that demonstrates the efficacy of our approach.

Type of Report: Other

The Design and Implementation of MCFlow: a Real-time Multi-core Aware Middleware for Dependent Task Graphs

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

***Abstract*—Modern computer architectures have evolved from uni-processor platforms to multi-processor and multi-core platforms, but traditional real-time distributed middleware such as RT-CORBA has not kept pace with that evolution.**

To address those issues, this paper describes the design and implementation of MCFlow, a new real-time distributed middleware for dependent task graphs running on multi-core platforms. MCFlow provides the following contributions to the state of the art in real-time middleware: (1) it provides an efficient C++ based 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; (2) it allows optimizations for inter-component communication to avoid data copying without sacrificing the parallel executability of data dependent tasks; (3) it strictly separates timing and functional concerns of an application so that they can evolve and can be configured independently; and (4) it provides a novel event dispatching architecture that uses lock free algorithms to avoid mutex locking and reduce memory contention, CPU context switching, and priority inversion. We also present an empirical evaluation that demonstrates the efficacy of our approach.

***Keywords*-component; middleware**

I. Introduction

Modern computer architectures have evolved from uni-processor platforms to multi-processor and multi-core platforms, but traditional real-time distributed middleware such as RT-CORBA has not kept pace with that evolution. For example, traditional real-time middleware requires explicit concurrency management and synchronization control which may scale poorly as the number of cores in a host increases.

MCFlow is designed to accommodate multiple real-time end-to-end tasks in a distributed system where some nodes have multi-core processors. Each end-to-end task can be represented as a directed acyclic graph (DAG). Schedulability analysis and subtask allocation is performed offline using graph-based analysis [1]. We focus on how to design and implement a middleware that can efficiently dispatch real-time end-to-end tasks in a distributed system with multi-core processors.

System Model: We consider a distributed real-time environment consisting of a collection of hosts. Each host can be a uni-processor or multi-processor, and the hosts are connected by a common network. A distributed real-time application implemented on MCFlow consists of multiple independent end-to-end tasks. Each end-to-end task is a collection of subtasks which forms an acyclic dependency graph. The subtasks can be freely distributed across hosts; however the allocations of host/core priorities for subtasks are statically bound at configuration time. Therefore, schedulability analysis must to be performed off-line to ensure feasible real-time performance of the system.

Task Model: We call a task that consists of a set of subtasks an end-to-end task, in which (1) subtasks are connected in a directed acyclic graph according to their precedence constraints and (2) deadlines of all sink subtasks are relative to the deadlines of their source subtasks. Given a subtask $T_{i,j}$ of an end-to-end task T_i, $Pre(T_{i,j})$ is the set of subtasks that are the immediate predecessors of $T_{i,j}$. If any of the subtasks in $Pre(T_{i,j})$ is located in a different host than the host which $T_{i,j}$ is located, $T_{i,j}$ is referred to as a *network triggered subtask*. On the other hand, if all of the subtasks in $Pre(T_{i,j})$ are located in the same node with where T_i is located, T_i is referred to as *locally triggered subtask*. If any of the subtasks in $Pre(T_{i,j})$ of a locally triggered subtask $T_{i,j}$ are located in a different core than the core in which $T_{i,j}$ is located, $T_{i,j}$ is also referred to as a *cross core subtask*.

The rest of this paper is structured as follows. Section II surveys related work on middleware and parallel computing. Section III describes challenges that motivate the design and implementation of MCFlow. Section IV presents a detailed discussion of MCFlow, including how it manages tasks across cores and hosts and how its configurable component model supports application development and deployment. Section V describes experiments designed to evaluate MCFlow and to validate its design and implementation. Section VI offers conclusions, and describes related work.

II. Related Work

The OMG Real-Time CORBA specification [2] supports network transparency for software component development and provides real-time policies and mechanisms including standard interfaces to specify resource requirements and configure object request broker (ORB) end-system resources such as thread priorities, message buffers, connections, and

network signaling, to control ORB behavior. TAO [3] is a full-featured Real-time CORBA [2] ORB. However, RT-CORBA is developed for single processor systems, and does not provide suitable mechanisms for QoS enforcement or optimization on multi-core platforms.

TAO's Real-time Event Channel [4] provides support for decoupled communication between objects. Instead of using point-to-point communication, interested event consumers subscribe for the types of events they need from the event service. The event service requires a centralized event dispatcher which may lead to high synchronization overhead and thus become a bottleneck on multi-core platforms.

The MC-ORB [5] middleware was specifically developed for multi-core platforms. However, it only tries to optimize the dispatching of network triggered tasks onto different cores, and does not consider how to optimize communication between cores.

Parallel programming languages, extensions, and libraries, such as Cilk [6], OpenMP [7] and Intel Thread Building Blocks [8] assume that the programmer should be responsible for exposing parallelism in source code but defer decisions about how to actually divide the work between processors to a run-time scheduler. However, these approaches do not provide suitable mechanisms for enforcing real-time constraints. Therefore, they are not suitable for real-time computing, nor do they provide direct support for environments that are both distributed and parallel. Dataflow programming languages and frameworks such as SystemC [9], StreamIt [10], and fastflow [11] also have been developed, but none of them is specifically designed to support real-time distributed applications where (1) computations can be flexibly configured for execution on a single core, between cores of a common host or between multiple hosts, while ensuring that (2) timing constraints such as end-to-end deadlines are strictly enforced.

III. Motivation

We consider an example in distributed real-time hybrid structural testing in which (1) one or more physical specimens are tested (especially with specimens in different sites due to experiment equipment constraints); (2) sensors measure the physical conditions of test specimens; (3) computation elements simulate numerical structure models; (4) a test coordinator and controller manage the experiment; and (5) hydraulic actuators generate movement of the specimens. Such an experiment may also include video streaming for safety monitoring. In this scenario, real-time guarantees are necessary when rate dependent physical elements are present in the test, or when coordination and synchronization of cyber elements is necessary for validation. It is natural to model the system as a set of end-to-end tasks with different rates and priorities. The computational elements in the example usually involve complex numerical computations that benefit from parallelization in order to meet timing constraints.

There are several important challenges that must be addressed in this example. First, the application is usually developed by civil engineers rather than computer scientists, so the software system should be easy to use. In current practice, this kind of application is usually developed with Matlab [12] and Simulink because their ease of use. However, Simulink models are not designed for distributed multi-core platforms and assume a purely time-triggered system. In our experience, purely time triggered systems cannot tolerate even small amounts of jitter from network communication.

Our previous work [13] [14] developed a specialized middleware that targets real-time hybrid testing. Like MCFlow, it is based on the data flow model; however, it lacks the ability to optimize for multi-core platforms. Achieving those optimizations across multi-core platforms is an important motivation for developing MCFlow.

IV. Middleware Design and Implementation

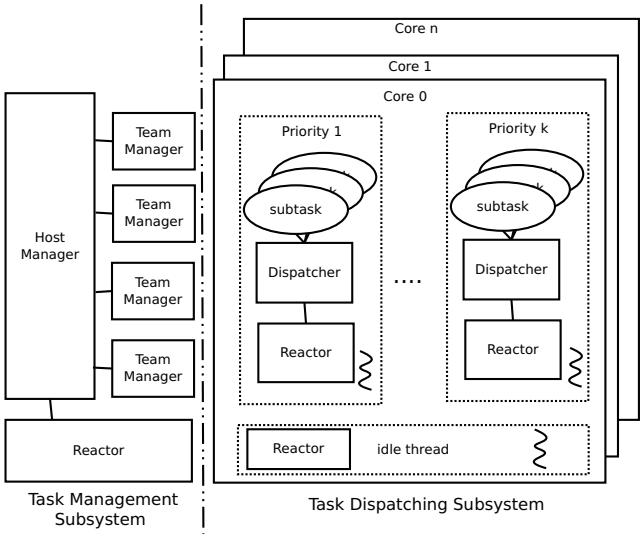

Figure 1. MCFlow Host Architecture

MCFlow enforces a crucial separation of concerns between its task management and dispatching subsystems, as shown in Figure 1. The task management subsystem creates, initializes and terminates *teams* of subtasks being allocated on the same host. In addition, it also manages any unrecoverable run-time exceptions of any subtasks by releasing all resources acquired by the subtasks.

Consistent with the real-time hybrid structural testing application domain [14] that first motivated the need for MCFlow, we assume here that real-time constraints only apply once a task graph has begun to execute, but not before them. For example, while a structural testing experiment is in progress timing guarantees must be enforced, but they are

not required while an experiment is being set up or taken down. Since the task management subsystem thus deals with non-urgent behavior of the system, only one thread (which runs at the lowest priority to avoid interference with real-time subtasks) is allocated for it. Although no core is explicitly designated for the thread so it can be migrated according to the operating system's scheduling policy, this thread could instead be pinned to a dedicated core and run at highest priority (e.g. to provide timing guarantees for task deployment as well as execution).

In contrast to the task management subsystem, the dispatching subsystem handles all real-time processing of subtasks. To enforce timing guarantees, all threads in the dispatching subsystem have higher priority than the task management subsystem thread. In addition, all thread resources including memory are strictly partitioned both for each individual priority and for each processor core. There are two reason for this partition. First, on multi-processor and multi-core platforms, the execution cost of thread migration can be unpredictable and can introduce significant delays [5]. By restricting each thread to a core, we can avoid the run-time overhead associated with thread migration. Second, memory sharing among threads may require extensive synchronization and locking control. 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. Duplicating the memory used by the middleware in each thread thus can effectively avoid the costs of thread migration and cache synchronization, at an additional storage cost that scales linearly with the number of cores.

We now describe the architecture of the MCFlow real-time middleware in further detail in the subsequent subsections. Section IV-A describes the task management subsystem, which prepares end-to-end tasks for execution. Section IV-B describes the dispatching subsystem, which enforces real-time execution of subtasks. Section IV-C describes MCFlow's component model. Section IV-D describes how MCFlow's component interfaces can avoid unnecessary data copying while preserving type safety. Section IV-E describes MCFlow's communication optimizations for network, inter-core, and intra-core data transmissions between tasks. Finally, Section IV-F describes how component configurations can be specified and realized in MCFlow.

A. Task Management Subsystem

As was mentioned in the previous subsection, the creation and destruction of subtasks in a host is managed by the task management subsystem. The host where an end-to-end task originates creates the subtasks assigned to it and issues initialization requests to other hosts in the systems. Upon acknowledgement from the downstream hosts, it activates the task dispatching subsystem to start real-time execution of the end-to-end task.

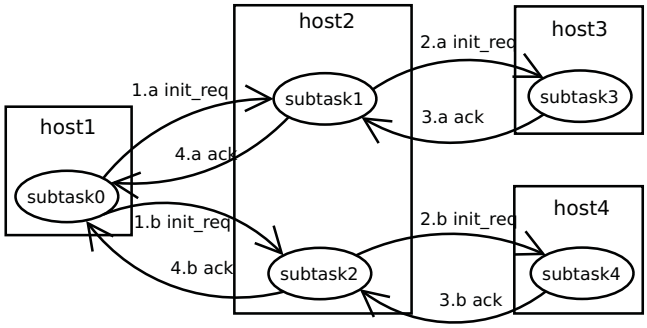

Figure 2. An example MCFlow initialization flow

If a host can create teams whose inputs originate from other hosts, it listens on a TCP socket and waits for incoming initialization requests. The host controller decides whether the request is legitimate and if so then creates concrete instances of the corresponding team manager and its associated subtasks. Subsequently, the request is passed along the dependency graph until all the downstream subtasks have been fully initialized before it sends acknowledgements back to the upstream subtasks. Figure 2 shows an example initialization process for an end-to-end task with 5 subtasks spanning 3 hosts.

Unlike the initialization phase for an end-to-end task which is entirely executed at the lowest priority, termination of each end-to-end task is partly executed at its real-time priority. This is to ensure that all the subtasks have stopped executing and can be safely deallocated. Whether a team is terminating can be controlled either by the data source or by run-time exceptions generated by the downstream subtasks. During the team termination phase, 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 along the task graph. In addition, it sends an asynchronous notification[1] to the team manager to indicate that subtask has stopped. Once the team manager receives all notifications from all subtasks in the team, it can safely deallocate all resources reserved for the team, at the lowest priority.

B. Dispatching Subsystem

The dispatching subsystem enforces the real-time execution of all subtasks in a host. Previous work [5] on dispatching real-time tasks used a half-sysnc/half-async concurrency architecture [15] for receiving network requests, in which a thread for each priority receives network messages and pushes requests into separate queues. A task allocation

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

thread retrieves the requests from the queues, interprets the requests, decides in which core and at which priority each request should be handled, and then pushes each request to a queue that is read by the designated thread pool for that priority in the designated core.

However, this approach suffers two context switches for each subtask that is remotely released from other nodes, which may impact timing guarantees especially with large numbers of subtasks. In this work, we assume that task allocation is done at configuration time. That is, given a subtask $T_{i,j}$, every subtask in $Pre(T_{i,j})$ knows in which node and core $T_{i,j}$ should be run, which is represented by a distinct network address and identifier. The network request is delivered directly to the appropriate core, rather than relying an extra subtask allocation thread to dispatch the request. This assumption allows us to do early dispatching at the network level without suffering the extra context switching seen in our previous work.

We have considered two approaches to realize this task dispatching scheme. In the first approach, each core uses one reactor to wait for all of its network requests. A thread pool uses the leader/followers pattern [15] to wait on the reactor with the highest priority. When a socket is ready for reading it is picked up by a thread, and the thread is then demoted to the corresponding priority for the socket before processing the request.

In the second approach, every core has several priority lanes, and each lane has a separate reactor with a dedicated thread (or thread pool). The first approach allows thread sharing among requests at different priorities, but incurs a small amount of priority inversion if a higher priority request becomes available while a lower priority request is still being processed. The second approach eliminates the possible inversion but may require that more threads be allocated in advance and also may consume more memory. However, our preliminary empirical study shows that the first approach suffers from higher cost and unpredictability when changing priorities by making Linux system calls. Therefore we chose to use only the second approach.

Besides network triggered subtasks, we also consider how to dispatch locally triggered subtasks. Dispatching locally triggered subtasks as though they were network triggered subtasks would incur unnecessary CPU overhead. An object collocation strategy is often used (e.g. in TAO [3]) to replace remote CORBA calls with direct local function calls whenever the caller and callee are in the same process.

However, without explicitly creating a new thread, using direct function calls would require callers and callees to be in same core. Therefore, another approach is needed when callers and callees may be in the same host but in different cores. We therefore integrate locally triggered subtask dispatching mechanisms directly into a reactor. Each reactor contains a local task queue, and the event loop in the reactor dispatches locally triggered tasks in the queue before it can be blocked in the event demultiplexing (`select()` or `epoll_wait()`) system call. When a subtask is complete and decides to release a successor task in another core, it enqueues the successor subtask into the corresponding task queue and then sends a signal to unblock the potentially blocking (inside `select()` or `epoll_wait()`) reactor thread.

To avoid race conditions in the event loop, we use the leader/followers pattern where only one thread at a time can pop a request from the task queue or be blocked in an event demultiplexing system call. Because our design ensures that all the threads having access to the same reactor must be executed in the same core, the serialization of the loop does not affect cache behavior or threads in other cores.

As shown in Figure 1, each priority in each core will have its own reactor, dispatcher and associated subtasks. MCFlow allows configuration of either one thread per reactor where subtasks can executed in FIFO order, or using the leader/followers pattern [15] by allocating multiple threads to wait on one reactor. The leader/followers configuration can be useful especially when subtasks from different end-to-end tasks are assigned to the same core and priority or when the subtasks can be blocked (e.g. by making certain system calls).

Each dispatcher manages a FIFO subtask queue and a timer queue to control when a subtask can be executed. When a subtask finishes its execution, it copies its outputs to the input queues of its immediate downstream subtasks and then inserts those subtasks into the subtask queues of their corresponding dispatchers. After that, it sends asynchronous notifications to their designated reactors. Once the associated thread of a reactor picks up the notification, it processes the notification in the following steps: (1) It pops a subtask from subtask queue. (2) It checks whether the popped subtask is still being processed (by reading an atomic flag); this step is required when the leader/followers pattern is applied because multiple threads may exist for the same core/priority and one thread may be still processing a subtask when another notification is sent by its upstream subtask and is picked up by another thread. (3) If the subtask is not being processed, it then checks whether the subtask is periodic and whether the release time for the subtask has expired. (4) If the release time has expired or the task is aperiodic, 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 on the reactor again.

Step 3 is required to enforce the release-guard semantics [1] across distributed or multi-processor systems so that *intervals between release times of jobs in any subtask are never less than the period of the subtask* [16]. Besides

waiting for the periodic boundary before a subtask can be executed, the release-guard protocol also allows a subtask to be executed earlier than the periodic boundary if the CPU becomes idle. To implement this feature, another *idle thread* with the lowest real-time priority (but higher than the priority used for task management) for each CPU waits for a prioritized reactor. 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 jobs 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, whenever it receives an idle notification.

The processing scenario is similar for network triggered subtasks. In this case, the subtask is notified directly from readability of the socket instead of asynchronous notification.

Although it is possible to use parallel programming technologies (such as Intel Threading Blocks [8] or OpenMP [7]) for locally triggered subtasks, those technologies do not provide a suitable common abstraction for network triggered subtasks, so that locally and network triggered subtasks must be programmed differently, making it more difficult to configure the allocation of tasks based on the results of scheduling analysis. Furthermore, those technologies use a central work stealing queue [6] for task dispatching, which is not suitable for real-time systems because a subtask in a queue can only be dispatched whenever a thread/core is idle. Even if a priority queue is used, if all threads are running lower priority subtasks, the higher priority subtasks in the queue won't be dispatched which results in a priority inversion.

Insertion into the FIFO task queue in each dispatcher needs to be properly synchronized because subtasks may be inserted from different cores based on the graph topology. Since there will only be one thread to consume tasks from the queue at a time, our implementation can be configured to use a mutex locked queue or a multiple producer single consumer lock free queue.

The input and output queue of each subtask is implemented as a lock free circular buffer. MCFlow ensures that no two elements in the circular buffer share the same cache line and thus avoids the *false sharing problem* [17] for the access to the buffer. The size of the buffer is configurable based on the period deadline ratio and the pipeline depth of the team, which can be calculated by the configurator.

C. Components

The behaviors of subtasks are defined in *components* which are written by application developers. A component in MCFlow is a C++ class that specifies its inputs, outputs, configuration parameters and runtime execution code. Conceptually, it is a function object with special hooks that determine how its input and output type should be initialized. Unlike traditional object oriented frameworks, MCFlow does not enforce any inheritance hierarchy on components but rather uses *interface polymorphism* based on template wrapper classes to encapsulate the components within subtasks that can be called directly by the dispatcher.

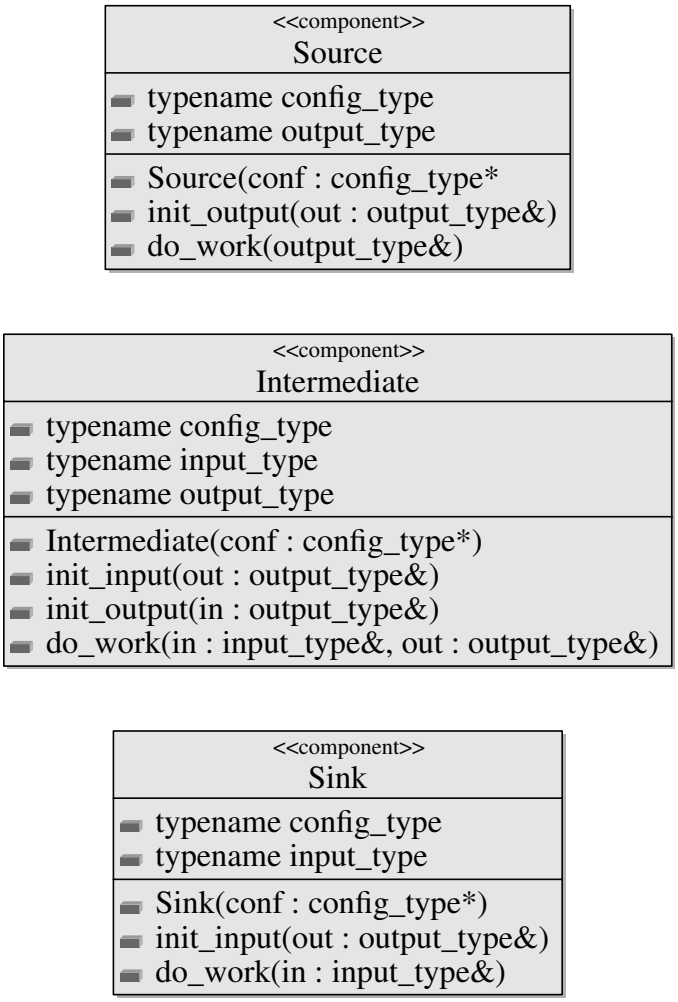

Figure 3. MCFlow Component Model

Components in MCFlow are classified into three categories (`Source`, `Intermediate` and `Sink`) depending on whether they generate output and/or consume input data as shown in Figure 3. Every component must provide an associated inner type called `config_type` and a constructor that accepts a pointer to its `config_type`. This allows users to control the initial states of components such as the maximum size of a matrix or certain parameters of a differential equation. The values of the configuration parameters will be provided by a configuration script as is

described in subsection IV-F.

D. Component Interfaces

MCFlow is designed to support both real-time performance guarantees and flexible component-based design. Unfortunately, dynamic memory allocation may introduce high cost and jitter, and yet to forbid the use of dynamic memory would 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 constructor. An application can usually utilize the `config_type` to set the size estimate at configuration time and get the value of the component constructor. However, as described in subsection IV-B, input/output buffers are implemented as cached aligned ring buffer to avoid synchronization and false sharing. If we put the input initialization code in constructors, those would need additional information about how many buffers need to be initialized and also would need to be sure to maintain memory alignment, both of which could complicate the interface. Instead, the separation of input/output initialization provides more room for the framework to optimize the input/output buffers without complicating the interface.

Program 1 shows an example of an intermediate component that computes a Fast Fourier Transform (FFT). Note that without optimization, a component designed in this way could lead to extensive memory movement among input and output buffers. There are two sources of such memory copying. First, an upstream subtask must copy its output result to the input buffer of its downstream subtask. Second, the downstream component must write to its own output buffer even if it can reuse its own input memory buffer. One solution to this problem is to define the input and/or output types as pointers instead of value types. However, that would requires both upstream component and downstream components to change their declarations to use pointers. Declaring the `input_type` or `output_type` to be a reference is not an option, because a reference type in C++ is not default constructible or reassignable; for example it could cause compiler errors if the framework tried to create an array of references.

To addres this issue, MCFlow provides a simple template wrapper class `ref_t` that encapsulates a pointer so that it is reassignable and can be implicitly converted to a reference type. In Program 1, if we changed the line `typedef vector<double> input_type` to `typedef ref_t<vector<double> > input_type`, memory copying from the upstream subtask to the downstream subtask could be avoided. If we also changed the `output_type` typedef and the second parameter of `do_work` to the `ref_t` type and write `output = input;` as the last statement of the `do_work` function, we can achieve the effect of in-place memory modification.

Program 1 Example Intermediate Component

```
struct FFT_Component
{
  struct config_type {
    int max_size;
  };

  using std::vector;
  typedef vector<double> input_type;
  typedef vector<double> output_type;

  FFT_Component(config_type* conf)
    : max_size_(conf->max_size){}

  void init_input(vector<double>& i)
    { i.reserve(max_size_); }
  void init_input(vector<double>& o)
    { o.reserve(max_size_); }

  void do_work(vector<double>& input,
               vector<double>& output)
    { ... }

  int max_size_;
};
```

Since we allow components 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. The major purpose for the sequence number is 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 end-to-end task, the output data of the first subtask won't be overwritten until the last subtask of the end-to-end 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 the pointer of 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.

E. Component Communication

One of the most important features of MCFlow is that it allows communication between components to be automatically optimized regardless whether it involves intra-core, inter-core or network communication. Communication is implemented by a set of template wrapper classes for the components. These wrapper templates are highly modular

and are specialized for different categories of components and their supported communication schemes. Table I shows the list of MCFlow wrappers for components. The `source_worker`, `intermediate_worker` and `sink_worker` are designed specifically for source, intermediate and sink components respectively. The `servant_worker` and `proxy_worker` templates are used for receiving and sending network messages. The `interthread_preparer`, `intrathread_preaperer` and `servant_preparer` listed on the first row of Table I are used to customize and potentially optimize how a component gets its input. For example, the C++ expression `intermediate_worker<FFT_Component, interthread_perparer>` represents a subtask which accepts input and produces output when none of the communication with its upstream subtasks is through the network. If the inputs were from the network instead of intra-host communication, we would change the second template parameter to be `servant_preparer`. Based on (1) the type of each component a wrapper template is designed for and (2) the allowed type of inputs for the component, Table I shows the valid combinations to wrap a component class as a dispatchable subtask.

F. Configuration Language and Code Generation

In order to make the type of communication between subtasks transparent to the application developers, MCFlow provides a tool which reads a configuration specification and generates C++ source files and makefiles. The content of a configuration specification includes:

- the hosts in the execution environment and their network addresses;
- all the end-to-end tasks for the system and their subtasks;
- in each subtask: the type of component used, the values for each field in the `config_type`, which host should the subtask be executed, and the priority of the subtask; and
- the connections between subtasks.

MCFlow enforces type safety for connections between subtasks. For example, given two components A and B, the connection from A to B is only valid if A.output_type is assignable to B.input_type. However, this potentially could limit reusability of components. In order to overcome this limitation, 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. Another useful feature of MCFlow connection is its *ports* which are essentially data members of the `input_type` or `output_type`. 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.

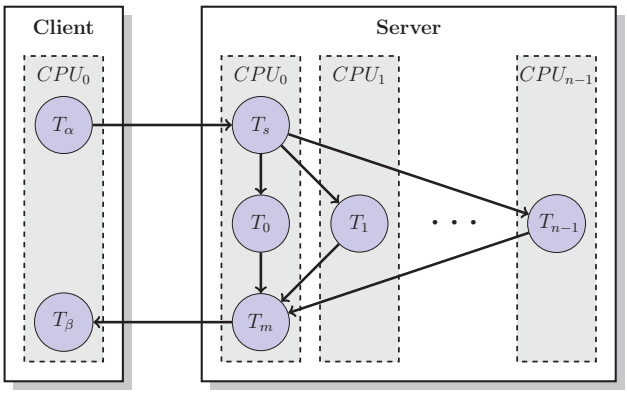

Figure 4. Overhead Experiment Setup

V. Performance Evaluation

The experiments described in this section were performed on a testbed consisting of two 6-core Intel core7 980 3.3GHZ CPU with hypertheading enabled. Both machines ran Ubuntu Linux 10.04 with 2.6.32 Kernel. 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.

A. Latency Comparison

We measure the latency of a traditional client and server scenario involving an end-to-end task with two hosts as shown in Figure 4. The server receives data from a client and splits the computations onto a number of CPUs, merges the result, and sends it back to the client. In order to evaluate how computation splitting affects performance, we vary the number of CPUs used. We use T_α and T_β to represent the data source and sink subtasks; T_s and T_m for the data splitting and merging subtasks; and T_i where $i = 0$ to $n-1$ for the split subtasks. The data transmitted between T_s to T_i, T_i to T_m is 64 bytes long. The data transmitted between T_α to T_s and T_m to T_β are $64n$ bytes long. No extra computation is done in T_α and T_β. The computation time for each subtask T_i is 5 μs and those of T_s and T_m are both $5n$ μs.

We compare the latency of these applications using MCFlow and TAO. The TAO version consists of two different configurations. The first configuration uses one ORB per CPU, with each ORB allocated only one thread. The thread is pinned on each CPU to avoid migration. All subtasks are assigned to their corresponding ORBs. The collocation strategy [18] 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

	interthread_preparer	intrathread_preaperer	servant_preparer
source_worker			
intermediate_worker	√	√	√
sink_worker	√	√	√
servant_worker			√
proxy_worker		√	

Table I
VALID PAIRS OF CONFIGURATIONS FOR MCFLOW COMPONENTS

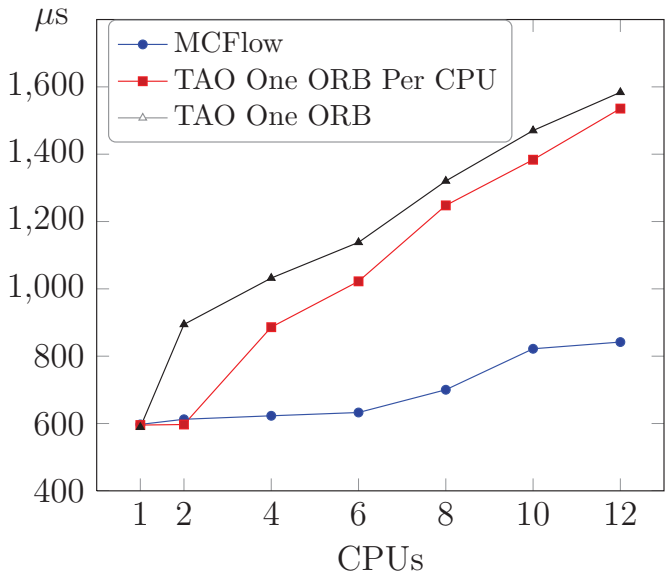

Figure 5. Latency Comparison Client Result

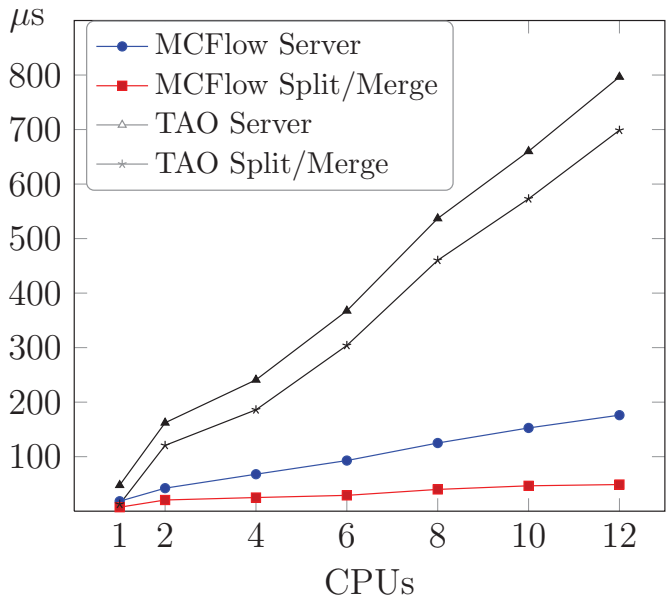

Figure 6. Latency Comparison Server Result

with the same ORB. The second configuration uses the leader/followers pattern with only one ORB per application and n threads. In this configuration, a subtask can't be run on a fixed CPU. No collocation optimization is used for this configuration; otherwise, all the CORBA invocations become normal function calls and thus the entire server would only run in one thread.

Figure 5 shows the time measured in the client from when T_α sent a request until when T_β receives a reply. Figure 6 shows the time measured on the server. The MCFlow server and TAO server curves measure the time from when T_s received a request to when T_m sends a reply. The Split/Merge curve measures the time from when T_s finishes it own computation, to when T_m receives the last request from any $T_i, \forall i = 0, \cdots, n$. For Figure 6, we only collect results for the per CPU ORB configuration, since in the single ORB version T_s and T_m may be in different CPUs which unless clock synchronization is used may involve potential measurement inaccuracy due to timing drift.

Figures 5 and 6 show little difference in performance when there is only one CPU. However, the latency for each TAO curve grows far faster than its MCFlow curve counterpart as the number of cores onto which tasks are split increases. This is largely due to the optimizations in MCFlow that can avoid data marshalling/demarshalling and/or socket communication between subtasks. 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 causes a lot synchronization overhead which leads to its poor performance. In contrast, the per CPU ORB version duplicates resources to each CPU and thus avoids such resource contention.

B. Real-time Performance

To evaluate the real-time performance of MCFlow on multi-core platforms, we designed the following experiment to examine whether MCFlow can preserve the priority constraints of an application.

In this experiment, we created three different end-to-end tasks: High, Med and Low. All the subtasks in High have higher priority than those of the other two end-to-end tasks; similarly, all subtasks in Low have lower priority than those of High and Med. Similar to the previous experiment, each

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

Table II
THE CPU AND WORKLOAD IN μS FOR EACH END-TO-END TASKS IN REALTIME PERFORMANCE EXPERIMENT

	High	Med	Low
50 Hz	0	0	0
60 Hz	0	0	0
70 Hz	0	0	0.059
80 Hz	0	0	0.168
90 Hz	0	0	0.329

Table III
DEADLINE MISS RATIO OF EACH END-TO-END TASKS WITH RESPECTS TO THE RATE OF LOW PRIORITY TASK.

end-to-end task spans two hosts and one host is used for the 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 end-to-end task is similar to Figure 4; however, different CPU assignments and workloads are used.

Table II shows the CPU assignment and workload of each subtask. The frequencies of High and Med are fixed at 200Hz and 100Hz respectively. We vary the frequency of the Low task and observe the effect of the Low task on the rest of system.

We assume the deadline of each task is equal to its period. The results of this experiment are shown in Tables III and IV. When the rate of Low is below 70, there are no deadline misses in the system. With an increase in the Low task's rate, deadline misses for Low increase. However, Low does not affect High or Med.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have described MCFlow, a middleware that can address the requirements for data dependent distributed real-time applications on multi-core platforms. It provides a simple model for component development in which an application developer does not need to worry about networking or data synchronization but rather uses a configuration tool to specify how components are connected and give their real-time constraints. Communication between components is optimized based on their location and topology and whether the communication is intra-core, inter-core or networking.

The data obtained from our experiments presented in Section V shows that the application of MCFlow performs similarly to TAO when only one CPU is used; however,

	High	Med	Low
50 Hz	4151	7375	9579
60 Hz	4170	7616	10884
70 Hz	4152	7478	10225
80 Hz	4151	7418	10167
90 Hz	4140	7519	10336

Table IV
AVERAGE RESPONSE TIME IN μS OF EACH END-TO-END TASKS WITH RESPECTS TO THE RATE OF LOW PRIORITY TASK.

MCFlow overhead is far less when multiple CPUs are involved. In real-time performance testing, MCFlow enforces the real-time constraint that higher priority end-to-end tasks are not affected by lower priority tasks.

As future work, we plan to conduct a more extensive study of how various constructs in MCFlow affect real-time performance of applications. We also plan to extend the support for various networking protocol and quality of service configuration parameters.

REFERENCES

[1] J. Sun, "Fixed-priority scheduling of end-to-end periodic tasks," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[2] Object Management Group, "RealTime-CORBA Specification, Version 1.2," November 2003.

[3] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill, "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, Feb. 2002.

[4] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time CORBA event service," in *Proceedings of OOPSLA*, Atlanta, GA, 1997, pp. 184–200. [Online]. Available: citeseer.ist.psu.edu/article/harrison97design.html

[5] Y. Zhang, C. Gill, and C. Lu, "Real-time performance and middleware for multiprocessor and multicore linux platforms."

[6] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995. [Online]. Available: http://dx.doi.org/10.1145/209937.209958

[7] 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.

[8] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

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

[10] S. Amarasinghe, M. I. 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, Jun 2005. [Online]. Available: http://dx.doi.org/10.1007/s10766-005-3590-6

[11] M. Aldinucci, S. Ruggieri, and M. Torquati, "Porting decision tree algorithms to multicore using FastFlow," in *Proc. 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.

[12] "Matlab - the language of technical computing," http://www.mathworks.com/products/matlab/.

[13] T. Tidwell, X. Gao, H.-M. Huang, C. Lu, S. Dyke, and C. Gill, "Towards Configurable Real-Time Hybrid Structural Testing: A Cyber-Physical Systems Approach," in 12^{th} *International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*. Tokyo, Japan: IEEE, Mar. 2009.

[14] H.-M. Huang, T. Tidwell, C. Gill, C. Lu, X. Gao, and S. Dyke, "Cyber-physical systems for real-time hybrid structural testing: a case study," in *ICCPS '10: Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*. New York, NY, USA: ACM, 2010, pp. 69–78.

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

[16] J. W. S. W. Liu, *Real-Time Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[17] 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*. Berkeley, CA, USA: USENIX Association, 1993, pp. 3–3.

[18] 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 *COOTS*. USENIX, 1999, pp. 145–160.