Supporting Deadline Constrained Distributed Computations on Grids

Xinghui Zhao and Nadeem Jamali

Department of Computer Science, University of Saskatchewan
176 Thorvaldson Building, 110 Science Place, Saskatoon, Canada, S7N 5C9
Email: {xinghuizhao, jamali}@agents.usask.ca

Abstract—The growing popularity of grid and cloud computing has led to a renewed interest in resource control and coordination. The Actor model, which encapsulates objects along with threads of execution, offers a convenient way for scheduling computations’ access to resources by way of scheduling of the actor threads. However, efficient Actor implementations do not use a thread for each actor, making implementation of fine-grained resource scheduling decisions difficult.

This paper presents our work on integrating mechanisms for deadline assurance into an optimized implementation of Actors. We achieve this by using deadline-driven adaptive scheduling, which prioritizes individual message deliveries and method executions involved in a distributed computation, based on the calculated deadlines by which each must be completed. These deadlines can be efficiently calculated at run-time for an important class of computations which use pipeline interaction style. Additionally, a tuner dynamically balances – manually or automatically – the overhead of the control mechanisms against the extent of control exercised. Experimental evaluation shows that the approach offers effective support for timeliness requirements (for multimedia QoS, for example) at the cost of a relatively modest and adjustable overhead.

Keywords—Adaptive scheduling; overhead tuning; actors.

I. INTRODUCTION

With the growing ubiquity of networked computers, there is an ever increasing potential for executing computations by utilizing distributed resources. As a result, computational paradigms such as grid and cloud computing become popular, where distributed applications can use resources available over the Internet. However, coordinating delivery of resources to distributed computations is a challenging problem. Because of the inevitable expiration of resources, the challenge is even more pronounced when the computations have timeliness requirements. Furthermore, open distributed environments can also add complexity to the problem. On the one hand, there is uncertainty inherent in both the evolving requirements of computations as well as the highly dynamic environment they would execute in; on the other hand, matching computations against resources has a high computational complexity [1].

In the Actors [2] model of concurrency, autonomous concurrently executing objects called actors, communicate with each other using buffered, asynchronous, point-to-point messages. By encapsulating objects along with threads of execution, actors offer a convenient way for coordinating resources among computations. Particularly, processor resources delivered to an actor can be controlled by appropriately scheduling the actor’s thread of execution [3]. However, it turns out that one-thread-per-actor implementation of actors is not particularly efficient; it is more efficient to have a pool of threads, where each thread processes messages for multiple actors, and the control flow of each actor is represented as a continuation [4]. This type of optimization has recently been shown [5] to deliver orders of magnitude improvement in performance for Actor Foundry, a Java library which faithfully implements Actor semantics. Specifically, for the Threading benchmark – in which 10 million messages are processed by 503 actors – an optimized version of Actor Foundry completes the benchmark computation in 10 seconds, where it took the unoptimized version 695 seconds to complete. More importantly, just this one optimization has led to Actor Foundry outperforming Scala [6], another Java-based Actor implementation which achieves good performance by compromising semantic properties of the standard Actor model, which took 12 seconds for Threading. In fact, it has brought Actor Foundry’s performance close to that of Erlang [7] (7 seconds for Threading), an efficient language which directly implements Actor semantics.

It turns out that the optimization which makes Actor systems efficient also makes coordination of resources between sub-computations more difficult. For instance, processor cycles can no longer be distributed between actors by simply creating a schedule which a processor scheduler can then enforce. The question we asked was: is this efficiency worth the lost ease of coordination? In other words, can coordination mechanisms be installed in the optimized Actor Foundry in a way that largely preserves the efficiency gains of the optimization? Particularly, we carefully examined the optimizing mechanisms for opportunities for control, and then tried to exercise it efficiently. In this paper, we present our efforts in this direction, which have shown that not only is the thread-of-execution level allocation of resources better suited for globally efficient fine-grained concurrency, its benefits can be largely preserved when supporting control mechanisms. Critically, installing control mechanisms in an implementation optimized for globally efficient fine-grained concurrency, although challenging to do, comes at a fraction
of the cost of having a separate thread for every actor.

As a first step towards effectively reasoning about and scheduling distributed resources, we have developed DREAM\(^1\) [8], a Distributed Resource Estimation and Allocation Model. DREAM uses resource terms to formally represent computational resources over time and space; these resource terms specify key attributes of the represented resources: type, density, time and location of existence. Computations are represented in terms of the resources they require. DREAM provides syntax and semantics for reasoning about future availability of resources, and about the feasibility of accommodating additional computations by the system. Next, we have designed a resource coordination mechanism based on DREAM, and installed the mechanism in the optimized Actor Foundry. Finally, a tuner has been implemented to – automatically or manually – control the overhead caused by the resource coordination mechanism. Our experiments have shown that significant coordination capability can be enabled by exploiting opportunities within the optimization mechanisms of Actor Foundry, for a relatively low overhead.

The rest of this paper is organized as follows. Related work is reviewed in Section II; Section III describes our coordination scheme which integrates DREAM into Actor Foundry, which is followed by a detailed description of the implementation in Section IV; evaluation methodology and results are discussed in Section V; and finally, Section VI concludes the paper and talks about some future directions.

II. Related Work

Both computational grids and clouds require distributed resource sharing: computational grids offer opportunities for bringing together disparate computational resources to solve relatively large problems; clouds allow users to lease resources/services – provided by service providers in the cloud – on a pay-per-use basis. Both computing paradigms face an important challenge: how to coordinate resource use by heterogeneous computations, especially when those computations have timeliness requirements. The performance of a such applications depends on how well the tasks involved in executing the application are matched against and coordinated on the available resources [9]. This is one of the main reasons for the growing interest in resource management and scheduling (RMS) in computational grids and clouds.

Traditional resource scheduling algorithms focus on minimizing the makespan of all jobs, and it has been shown that mapping jobs onto heterogeneous resources while minimizing makespan is an NP-complete problem [1]. A variety of heuristics [10] are therefore adopted to schedule jobs on grids. However, as the diversity of applications using computational grids and clouds grows, the resource requirements which need to be satisfied are also becoming more complex. Minimizing makespan is no longer the only requirement for a scheduling scheme. QoS requirements offer an instructive example, where resources are required at a certain time, for a certain period, and often there is a need for multiple types of resources to be available together. To satisfy such QoS requirements, more sophisticated and finer-grained resource coordination mechanisms are required.

There are a number of architectures which support QoS control over grids. GARA (Globus Architecture for Reservation and Allocation) [11] is one of the earliest architectures for supporting QoS over heterogeneous resources. G-QoSM (Grid Quality of Service Management) [12] is a service-oriented QoS management model which provides policy-based admission control for resource reservation. AssessGrid [13] provides QoS support by assessing and managing the risk (probability of failure) of accommodating new jobs. All of these are reservation-based frameworks, which require applications to explicitly specify when they require resources, information based on which reservations are made. Unlike these approaches, our work offers resource reasoning ability which can automatically generate a resource reservation schedule for each application, such that its deadline constraint can be satisfied. Applications only need to specify the deadlines by which their computations should be completed.

Another class of approaches for resource management on grids and clouds includes different types of economic models, which are especially popular in cloud computing because of the pay-per-use emphasis in clouds. In such economic models, resources are organized in markets, and users who require these resources can pay to have them. Decision making about resource allocation in these models is typically driven by common market mechanisms, such as auctions and negotiations. However, Market-based approaches are often criticized for their performance because of their high overhead, which is often caused by failed negotiations/allocations, or low occupancy of resources. In [14], Chard et. al. have studied resource utilization strategies which aim to reduce allocation failures, increase occupancy and hence increase the performance by using techniques such as overbooking, advanced reservation, etc. In [15], Chang et. al. address the performance issue from a different perspective: instead of increasing occupancy, i.e., using as many available resources and as efficiently as possible, their approach tries to use as few resources as possible to reduce cost and administration overhead for setting up the resources. We deal with overhead in a different way. Instead of measuring the performance of the system as a whole, we measure the extra overhead caused by the resource allocation mechanism itself, and set it as a parameter which can be tuned by users. Particularly, we provide users the ability to explicitly manage the overhead, which in fact represents resources taken by reasoning about resource availability and generating schedules for applications.

---

\(^1\)Called ROTA (Resource Oriented Temporal Logic for Agents) in our previous work.
III. INSTALLING A RESOURCE CONTROL MECHANISM

To be precise about what we mean by distributed computations, we take them to be actor computations which are spread over a distributed execution space. Actors are the de facto model of concurrency underlying a number of languages, e.g., Scala [6], Erlang [7], etc. While a number of these are Java library implementations, Erlang [7] is directly implemented. Despite the advantage that Erlang enjoys for being directly implemented, being the most efficient Actor implementation, it is of special interest because it sets a performance standard by which other implementations can be compared. However, for multiple reasons, of even greater interest to us is Actor Foundry. Most notably, Actor Foundry attempts to faithfully implement the Actor model, which is well understood, its code is uniquely accessible because of its modular design, and finally, a simple optimization is known to bring its performance close that of Erlang’s. For these reasons, we use Actor Foundry for our prototyping.

Because delivery of processor cycles is essentially through the scheduler, the focus of our attention is Actor Foundry’s scheduling mechanism. AF’s scheduler schedules a fixed number of native JVM threads called workers, which in turn select from among the actors waiting in a waiting queue to get a chance to execute. The waiting queue is shared by the worker threads. An actor waits in this queue only when it has received a message in its own message queue; at all other times, it is essentially dormant, and does not need to execute. Whenever a worker becomes free, it picks an actor from the waiting queue to execute. The waiting queue is a FIFO queue. Actors are placed in the queue according to the order in which they received the first messages in their message queues. Although an actor’s deadline for processing messages is in its queue has been processed. After completing with one actor, the worker dequeues another actor and starts to execute it. We can say that Actor Foundry’s scheduler is message-driven in the sense that only actors which have received messages in their queues get to be on the waiting queue; any other actors stay off of it. It is obvious from the way that Actor Foundry schedule actors, that it cannot support timeliness requirements of computations. However, we show that replacing the FIFO scheduler with an adaptive scheduler, one which can alter the order in which actors can be scheduled based on computations’ deadlines, can support timeliness requirements.

Fundamental to our approach is an identification of finer grained actions to which deadlines apply. In other words, although a multi-actor distributed computation may have a deadline by which it needs to be completed, that in itself is too coarse grained an information to enable effective control.

On the other hand, too exact a scheduling would be too costly in terms of scheduling overhead. It turns out that there is a middle ground where scheduling granularity is just fine enough that it offers sufficient control for a variety of applications. This is what our evaluation will eventually seek to illustrate. The granularity of control we aim for is at the level of an actor’s processing of a message. In other words, we do not pre-empt an actor’s method execution once it has begun.

The deadline for an entire (typically multi-actor) computation is specified by the user / programmer, indicating the time by which the computation ought to be completed. Individual actors participating in the computation have deadlines by which they must finish processing all messages in their individual queues. The deadline for an actor participating in a computation defaults to the entire computation’s deadline. A more interesting type of the deadline – the one which determines the granularity of control offered by this approach – is the deadline for processing of individual actor messages. The deadline for an actor to process one of its messages can be computed by subtracting from the actor’s deadline for completing all of its messages, the time required for subsequent computation required for processing its remaining messages. For a class of applications, these deadlines can be efficiently calculated by counting back from the entire computation’s completion deadline, accounting for the time required for processing each message (i.e., executing the method required for processing it). This is discussed in greater detail in Section III-A.

Once the deadline for a message has been computed, the deadline is tagged onto the message by the runtime system, to be subsequently read by the recipient actor’s runtime (which may be the same as the sending actor’s runtime) and used in deciding when to schedule delivery of the message to the recipient, and consequently, when to schedule execution of the actor. Particularly, resource terms specifying available resources are matched against (multi-actor) computations’ requirements – represented by the deadlines and the code to be executed by those deadlines – to identify computations which can potentially be accommodated given the available resources. The matching process, which uses DREAM [8], also generates a (possibly distributed) resource allocation schedule for each computation. The enforcement of this schedule is attempted by manipulating the order in which actor messages are processed.\(^\text{4}\)

\(^2\)The number of worker threads is increased at run time when found to be insufficient for progress in the computation.

\(^3\)We make a simplifying assumption that an actor continuously participates in a single computation until the computation is completed. To generalize, we would need to track messages by the computations they belong to, and an actor would have deadlines associated with each computation it participates in, and then have separate deadlines for completed processing of all messages associated with each computation.

\(^4\)Modifying processing order of messages does not violate Actor semantics. As long as messages are eventually delivered, the fairness requirement is satisfied.
A. Deadline Analysis

Calculating deadlines for messages is critical for providing fine-grained QoS control for individual actors. In general, the deadline for any message can be calculated using a known deadline for the recipient’s subsequent deadline and counting back by the amount of computation and communication required to be carried out before that deadline must be met. This obviously is non-trivial for a computation with sufficiently complex interaction between actors. However, for classes of computations – such as those which use the pipeline communication style – this can be efficiently achieved. Examples of such computations include multimedia delivery, as well as concurrent algorithms to solve a wide variety of programs using the pipeline style of interaction. We count back from the entire computation’s deadline to determine message-grained deadlines; this can be done in time linear in the total number of messages. In fact, as is the case with the video-conference example we discuss later, often the pattern is regular enough that it is a matter of tracking the order of messages, which eliminates the need for computing down to the first message, allowing the deadline-computation to proceed alongside the actual computation.

B. Extending Actor Foundry

We extend the Actor Foundry framework by integrating a deadline reasoning component into the scheduler, and add a Tuner facility to observe and adjust the ratio of resources consumed by computations and the reasoning mechanism. Figure 1 shows the architecture of the modified scheduler.

![Figure 1: Integrating Deadline Reasoning into Actor Foundry Scheduler](image)

First, we modify Actor Foundry’s essentially message-arrival driven scheduler so that it becomes a deadline driven scheduler. In other words, we replace the FIFO queue with an Earliest Deadline First (EDF) queue. Recall that we are interested in computations which have predefined deadlines, by which they are expected to be completed. When there are multiple actors executing as part of a computation, by default, each actor is thought to have the same deadline as that of the entire computation. All actors that are waiting for execution are placed in a priority queue according to these deadlines. In other words, the actors in the system are scheduled on a Earliest Deadline First (EDF) basis, which has been shown to be the optimal scheduling algorithm on preemptive uniprocessors [16], in the sense that if a set of real-time jobs can be scheduled by any other scheduling algorithm, it can also be scheduled by EDF.

Second, we use a special meta actor to carry out resource reasoning using the DREAM model. This DREAM Reasoner (DR) is responsible for carrying out the following tasks:

- For each asynchronous message in the system, determine the deadline by which the processing required by the recipient actor (i.e. processing of the message) has to be completed.
- Calculate if the deadline of a message can be met using available resources. If so, place the message in its recipient actor’s mailbox – implemented as a priority queue as well – on an earliest deadline first basis.
- Update resource availability of the system.5

Third, we use a Tuner to perform meta-level resource control. Because the reasoning mechanism itself consumes computational resources (amounting to the overhead), this tuner offers a means to balance the division of resources consumed by the computational actors and those consumed by the reasoner. This is possible to do trivially on a single processor because of the way in which we have implemented the reasoner as a separately scheduled meta-actor. The tuner can be fixed at a particular division of resources, it can be set to automatically react to observed progress of the computation, or it can be made available to a system operator in the form of a tuning knob. In a fixed setting, the ratio between the processing power taken for the computation vs. that for the reasoning (e.g. 80%:20%) can be initialized at the beginning of the computation. Section III-C discusses one way in which the tuner can set itself reactively.

Installation of these mechanisms leads to three levels of control: coarse-grained actor scheduling, fine-grained message scheduling, and finally, a meta-level division of resources between the reasoner and the actual computation being reasoned about.

C. Self-Tuning

Consider an adaptive tuning policy where the meta scheduler begins by maintaining a 20%:80% ratio between resources for the reasoner versus the computation, but is allowed to increase the resources for the reasoner to a maximum of 30%. Particularly, let us say that we are

---

5 Available resources are represented by a set of resource terms, which are updated at run time as the resources get allocated to actors or they expire. Resource terms continually expire as the intervals of their definition pass, and when they do, they can be garbage collected. This garbage collection is important for efficient resource reasoning because the cost of reasoning has a linear relationship with the number of terms describing the current state of resource availability.
supporting a computation where the reasoning can happen alongside the computation (such as the video conference application to be introduced later)\(^6\), and the adjustment in the division of resources happens in response to specific triggering events. Consider three type of triggers. First, \textit{reasoning too slow} event is triggered when reasoning about a message is taking so long that it is not completed by the time a decision is needed. This trigger results in a slight increase in the ratio of resources provided to the reasoner. On the flip side, \textit{reasoning too fast} event is triggered when it is observed that reasoning about a number of messages has completed much sooner than the decisions were actually needed. In this case, a greater portion of the resources can be diverted toward the actual computation. Finally, a third event \textit{reasoning too costly} is triggered if despite the threshold of 30\% for reasoning being reached, the reasoning is not adequately fast for arriving at decisions in time for when they are needed. This could happen if the the reasoning requires more than 30\% of the time, but the deadlines are so tight that there simply is not enough room to fit in that much reasoning. One reaction to this event is to simply give up on reasoning, and simply devote all resources to the computation itself, hoping for the best.\(^7\)

IV. IMPLEMENTATION

A prototype implementation has been developed by extending Actor Foundry. Actor Foundry supports distributed computations by supporting actors at a number of nodes, enabling communication between actors across node boundaries, and by supporting actor migration.\(^8\) The architecture of an instance of AF on one node is shown in Figure 2.

![Figure 2: System Architecture (AF Node Instance)](image)

The \textit{Actor Manager}, together with the newly added component, \textit{DREAM Reasoner} (DR), serve the core runtime functions of a foundry node. They are responsible for creating and scheduling new actors, as well as handling messages between local actors, and moving messages between the local foundry node and other foundry nodes. When the first actor is created as part of a computation – typically how a multi-actor computation is initiated – its deadline is set to be the computation’s deadline, as specified by the user. Any actor subsequently created by an actor is assigned the same deadline as its creator’s.

Whether the resource reasoning is enabled or not can be decided at the time of initiating the computation’s execution. If the resource reasoning is enabled, every actor message which involves the local foundry node – specifically, an entirely local communication, a message from a local actor intended for a remote actor, or a message for a local actor from a remote actor – is examined by the DR as a reasoning request. Upon receiving a reasoning request, the DR calculates its deadline (as needed), allocate local resources if the message is to be processed locally, or dispatches it if its destination is a remote actor (for the allocation to subsequently happen at the remote location).

Algorithm 1 shows how the system generates a resource allocation for processing of an actor message at the location of an message’s recipient. These allocations are made for each of the sequences of actions requiring no change in the type of resource required. We refer to these sequences of actions as \textit{segments}.\(^9\) The input parameters of the algorithm, therefore, are: available resources in the form of resource terms; and the resource requirements for processing each segment of the method corresponding to the message. The algorithm looks to accommodate the message using available resources, and returns a schedule for the message in the form of resources being reserved for it, or \textit{null} if a feasible schedule is not found.

The complexity of Algorithm 1 is \(O(s \times t \times r)\), where \(s\) is the number of segments of continuous resource use per message, \(t\) is the number of resource types each segment requires,\(^10\) and \(r\) is the number of resource terms. In any real application, \(t\) is usually a small constant; \(s\) too is typically a small constant, but depends on the granularity of control desired. In other words, the complexity is typically \(O(r)\).

The number of resource terms \(r\) during the course of reasoning depends on the fragmentation of resources. There are a number of ways in which we can improve on this complexity. Recall that resource terms are defined in time and space. This means that only a small subset of the resource terms will be relevant to a computation which needs

---

\(^6\)In fact, this mechanism was used in the implementation for which we carried out our evaluation.

\(^7\)In ongoing work, we have also tried reacting to this trigger by reducing the frequency of reasoning. In other words, we decide deadlines for only a certain percentage of messages (say, every other).

\(^8\)This is made possible by the fact that actors have globally unique names, with mapping between the names and actual physical locations tracked using distributed name tables.

\(^9\)Specifically, a sequence of actions, each requiring the same single type of resource, can be combined into one segment; an action which requires multiple types of resources is considered as one segment, because these types of requirements must be reasoned about separately from others to assure simultaneous availability of multiple resources.

\(^10\)Here we mean something specific by types: the same kind of resource (say, processor) at two different nodes is considered two types. To be more precise, we use the term \textit{located type}. 

Algorithm 1 AccommodateMessage(terms, requirements)

1: schedule = null
2: start = end time of the previous message accommodated
3: end = start /* starting from the earliest start time, try to
find end time of the message */
4: s = number of message segments
5: for i = 1 to s do /* sequentially accommodate message segments */
6: t = number of resource types in requirements[i]
7: for j = 1 to t do /* reason about multiple types of
required resources separately */
8: reserved = null /* record reserved resources for the
segment */
9: r = number of resource terms
10: for k = 1 to r do /* traverse resource terms */
11: select terms during (start, deadline), in resource
type requirements[i][j] /* pick all resources
of required type which exist before message’s
deadline */
12: end for
13: look for time instant end such that portion of
selected terms defined during (start, end') =
requirements[i][j] /* try to allocate sufficient
resources to complete execution of segment */
14: if end does not exist then /* there is not enough
resource (type j) available before the message’s
deadline */
15: cancel reservation, return null
16: else /* enough resource is found */
17: reserve selected terms during (start, end')
18: schedule += reserved /* add reserved resources
to schedule */
19: if end' > end then
20: end = end' /* update the end time of the
segment */
21: end if
22: end if
23: end for
24: start = end /* set up the earliest start time of the next
segment */
25: end for
26: return schedule

to be carried out during a time interval (defined by the
earliest start time and deadline). A way of filtering the set
of resource terms for an interval of time will significantly
reduce the number of resource terms to be considered.
Also, the most fragmentation is likely to happen in the
nearest future. This can be good and bad. It is bad because
computations which need to be accommodated in the near
future have to contend with reasoning involving a larger
number of resource terms; it is good because the resource
terms in the near future will expire once their time of
existence has passed. In ongoing work, we are looking at
ways to control this complexity by managing this fragment-
tion. Most interestingly, we are looking at another tuning
opportunity involving the distance in the future to look into
to accommodate a sufficiently permitting computation. In
other words, one way to balance the resources devoted to
reasoning versus the actual computation would be to decide
how far in the future to look for the needed resources.
We envision a tuning knob for this type of meta-level
control very similar to the one described previously for
balancing ratio of processing devoted to reasoning versus
the computation.

Actor Foundry’s scheduler has been rewritten to accom-
mmodate DREAM Reasoner. A dedicated thread is used for
carrying out DREAM reasoning. The scheduler schedules
the DREAM thread and the collection of worker threads
in turn according to the ratio set through the tuner. Actor
Foundry’s implementation of actors is essentially unchanged
in our extension except for the fact that there is a deadline
associated with each actor’s completion.

V. EXPERIMENTAL RESULTS

Our approach to evaluating this solution is to illustrate that
deadline support does not have to come at the expense of
global efficiency. In other words, the adaptive scheduling
mechanism we have installed into Actor Foundry, which
limits itself to doing a responsive reordering of message
deliveries and actor execution orders, can achieve timeliness
goals of an important class of applications without incurring
a heavy overhead.

Two sets of experiments were carried out to illustrate the
effectiveness and efficiency of this approach. The first
set used a deadline constrained version of a benchmark
computation, Threading, originally developed for evaluat-
ing efficiency of actor implementations. The second set
of experiments examined application of our approach for
synchronization of audio and video streams for a live video
conferencing.

A. Threading Benchmark with Deadline Constraints

Threading is a benchmark developed by Karmani et
In Threading, a number of actors pass a token a specified
number of times. Here in our experiments, we created 10
Threading computations with specified deadlines, where in
each computation, 503 actors pass a token for 2000 times.
We set the deadlines to be \(i \times t + \Delta_i\), where \(i\) is the id
of the Threading \((1 \leq i \leq 10)\), \(t\) is the approximate
time required for completing a Threading computation,
and \(\Delta_i\) is the a small number representing the tolerance
for each computation. Note that we set these deadlines as
tight deadlines to illustrate the efficiency of the approach.
The experiments were run on a MacBook Pro laptop with
Intel Core Duo CPU @ 2GHz, 2GB RAM and 2MB L2 cache. The value used for \( t \) was 280ms, and \( \Delta_i \) is randomly generated within range \([0, 200\text{ms}]\).

As shown in Figure 3, only one out of ten deadlines was missed for our approach, Actor Foundry with deadline support (AF-D), where (predictably) AF misses 9 of the deadlines. Obviously, AF does not claim to support deadlines; the comparison with AF here is simply to establish a baseline. The additional overhead of using AF-D (i.e., the cost of supporting the deadlines) included a startup overhead of approximately 60ms, followed by an average of 5ms for each Threadring execution completely over the period of 280 ms. Discounting the setup cost, the additional overhead of AF-D amounted to 2% above the cost of carrying out the computation using AF.

B. Live Video Conferencing

Multimedia applications are characterized by their Quality of Service (QoS) requirements, such as low jitter, inter-stream skew, etc. A set of experiments examined application of our approach to QoS for live video conferencing. Unlike existing domain-specific approaches to multimedia QoS, we apply our general approach to address the problem. We chose live video conferencing to illustrate the effectiveness of our approach, because the application requires very low end-to-end delay (<150ms), as well as audio/video inter-stream skew (<100ms).

We implemented the Real-time Transport Protocol (RTP) [17], which is designed for end-to-end, real-time transfer of stream data. In our setup, microphone and camera actors sample sound and images at the speaker end and encode them into audio and video packets, which are transmitted to the audience end over a network. The ratio of audio and video packets of 1:10 is approximately the same as in RTP. At the audience end, these packets are decoded and played by the audio player and video player actors in sync.

We compared synchronization and its computational overhead when using our system AF-D with the original Actor Foundry. For AF-D, length of the session (i.e., deadline for completion of entire computation) and acceptable lag were specified; otherwise, the program remained unchanged. AF-D automatically calculated fine-grained deadlines for delivery of each packet and then synchronized the delivery of those messages according to their deadlines. The system was set up to be self-tuning.

In the experiments, for source node (speaker side), we used a MacBook Pro laptop with Intel Core Duo CPU @ 2GHz, 2GB RAM and 2MB L2 cache; for destination node (audience side), we used a Dell XPS laptop with Intel Core Duo CPU @ 2GHz, 3GB RAM and 2MB L2 cache.

As shown in Figure 4, in our experiments, AF-D satisfactorily synchronized delivery of audio and video packets. The maximum audio/video inter-stream skew was 78ms; below 100ms is considered acceptable. The overhead incurred was 53ms over the course of a computation requiring 1607ms of execution time, which is approximately 3.5%.

Results from these two sets of experiments illustrate that our approach, although constrained by the opportunities provided by relatively coarse-grained scheduling changes, nevertheless offers a degree of control that is sufficient for applications with timeliness constraints such as Quality of Service requirements.

VI. CONCLUSION AND ONGOING WORK

With growing popularity of computational grids and clouds, efficient coordination of resource sharing between computations is becoming increasingly important. Where this is particularly true for applications such as multimedia with QoS requirements, computations with less demanding implicit timeliness requirements executing in the cloud also stand to lose as a result of resource uncertainty.
The marriage between global efficiency and fine-grained coordination is often a difficult one. In this paper, we presented how we installed control mechanisms in an Actor implementation by exploiting opportunities to change the orders of actor execution and message delivery. Despite the course grain of these opportunities, our experimental evaluation shows that the approach can be effectively used for satisfying deadlines in important classes of applications with timeliness requirements, and with relatively low overhead.

We are following up on this work in a number of directions. First, although we have shown that if the fine-grained deadlines can be efficiently calculated, the reasoning and subsequent scheduling is efficient, it is not clear the deadlines can always be efficiently calculated. We are looking at identifying a wider class of computations – defined by the communication styles they use – for which these fine-grained deadlines can be efficiently calculated. A promising opportunity may also lie in exploring whether programmers can easily provide hints about the type of interaction in the computation. For example, in the live video conference application, intimate knowledge of the communication style enabled us to compute message deadlines as we progressed in the computation instead of computing them in advance. It would be interesting to know if such opportunities exist for other communication styles, and if so, how to incorporate programmer input.

Second, we are extending the work on the tuner by exploring different dimensions along which we might try to balance the use of resources between reasoning and carrying out the computations. One promising opportunity appears to lie in controlling how far in the future the reasoning mechanism looks in search of needed resources.

Third, we are using the resource schedules generated as a result of this type of reasoning to exploit opportunities to conserve energy by adapting hardware operation. Specifically, if computations come with explicit deadlines by which they should complete, then they need not complete any sooner. This creates an opportunity to operate fewer processors and to operate necessary processors at the lowest required frequencies for satisfying the computations’ requirements. We are working to produce application-adaptive energy efficient schedules for operation of hardware.

REFERENCES


