Adaptive Scheduling and Overhead Tuning for Deadline Constrained Computations

Xinghui Zhao and Nadeem Jamali
Department of Computer Science, University of Saskatchewan
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 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. This paper presents our work on integrating mechanisms for deadline assurance into an optimized implementation of Actor Foundry. We achieve this by using deadline-driven adaptive scheduling, which prioritizes individual message deliveries and method executions involved in a distributed computation. Additionally, a tuner dynamically balances the overhead of the control mechanisms against the extent of control exercised. Experiments 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—Overhead tuning; adaptive scheduling; actors.

I. INTRODUCTION

Coordinating delivery of resources to distributed computations is a challenging problem. On the one hand, there is uncertainty inherent in both the evolving requirements of computations as well as the highly dynamic environment they execute; on the other hand, matching computations against resources has a high computational complexity [1].

In the Actor [2] model of concurrency, autonomous concurrently executing objects, namely 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. However, one-thread-per-actor implementation of actors is not particularly efficient; it is more efficient to have a pool of threads executing actors in some order. This type of optimization has recently been shown [3] to deliver a performance for Actor systems that comes close to the performance of Erlang [4] (10 seconds for Actor Foundry [3] – a faithful implementation of Actors – vs. 7 seconds for Erlang for the Threading benchmark).

Our work shows that 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.

II. INSTALLING RESOURCE CONTROL MECHANISMS

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 obviously 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. For Actors, it turns out that message processing grain represents a good compromise.

Calculating deadlines for messages is key to providing fine-grained QoS control for individual actors. The deadline for a message can be calculated using a known deadline for the recipient’s subsequent deadline and counting back by the amount of computation required to be carried out before that deadline must be met. This is non-trivial for a computation with sufficiently complex interactions between actors. However, for a class of computations – which use the pipeline communication style – this can be efficiently achieved. Examples of such computations include multimedia delivery, as well as pipeline versions of algorithms for solving a wide variety of problems. The count back from the entire computation’s deadline 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.

Once fine-grained deadlines have been determined, resource terms specifying available resources can be matched against (multi-actor) computations’ requirements to identify computations which can potentially be accommodated given the available resources. The matching process, can then generates a (possibly distributed) resource allocation schedule for each computation, which can finally be enforced by manipulating the order of delivery of actor messages.

Figure 1 shows how we have installed these mechanisms in Actor Foundry. First, we have replaced Actor Foundry’s essentially message-arrival driven scheduler with a deadline driven scheduler. Second, a special meta actor has been installed to carry out resource reasoning based on the DREAM model [5] (called ROTA in previous work). Third, a Tuner facility has been installed to perform meta-level resource control. 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
III. EXPERIMENTAL RESULTS

Our evaluation seeks to show that the adaptive scheduling mechanism, 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. A set of experiments examined application of our approach to Quality of Service (QoS) for live video conferencing. Unlike existing domain-specific approaches to multimedia QoS, we apply our general approach to address the problem.

We implemented the Real-time Transport Protocol (RTP) [6], 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.

As shown in Figure 2, 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%.

IV. CONCLUSION AND ONGOING WORK

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.

Follow-up work is ongoing in a number of directions. First, 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. Another opportunity may lie in exploring whether programmers can easily provide hints about the type of interaction in the computation. Second, we are exploring different dimensions along which we might be able to use a tuner to balance the use of resources between reasoning and carrying out the computations. One possibility is to control how far in the future the reasoning mechanism looks while searching for needed resources. Third, we are working on using the resource schedules generated as a result of this type of reasoning, as a basis for making decisions about hardware operation for conserving energy.

REFERENCES