Artificial Deadlock Detection and Correction in Bounded Scheduling of Process Networks

by

Basu Vaidyanathan

Term Project

Embedded Software Systems

(EE382C)

The University of Texas at Austin

December 1999

ABSTRACT

Parks devised a scheduling policy for process networks that simultaneously satisfies both complete execution and bounded memory execution requirements of any dynamic scheduler. This paper presents an implementation of dynamic artificial deadlock detection and correction in Parks bounded scheduling of process networks. The presence of artificial deadlock when atleast one process is suspended attempting to write to a full queue, is detected before it actually happens and is resolved so that there is a steady progress in the execution of process networks program with bounded memory whenever possible. This paper describes a way to completely eliminate artificial deadlock thereby speeding up the program execution with bounded memory.

INTRODUCTION

A process network model is a model of computation in which concurrent processes interact through one-way first-in first-out (FIFO) queues. A process network is a natural model for signal processing systems that deal with infinite streams of data values, to realize concurrent processing of functional parallelism. Ptolemy is a simulation and prototyping environment that uses process network model.

Though true real-time measurements are difficult in the simulation of digital signal processing applications, process networks computation model helps develop a prototype on workstations that significantly reduces cost and development time over an equivalent hardware implementation. A real-time sonar beamformer [2] has been successfully simulated in software using process network model on multi-processor Unix workstations interfacing with POSIX lightweight threads library. Currently, this implementation does not handle artificial deadlock; it rather circumvents it by choosing a fairly large queue size.

Our implementation addresses this problem and attempts to detect artificial deadlock before it happens and corrects it so the process networks program could make steady forward process with bounded memory whenever possible. It is not very difficult to realize this implementation enhancement, however, it is quite involved due to the complex behavior of process networks program.

BACKGROUND ON PROCESS NETWORKS AND BOUNDED SCHEDULING

Kahn process networks is a computation model in which each process produce data elements called tokens on FIFO queues of infinite length. In this model, the execution of the process is suspended if it attempts to consume data from an empty queue. A process may not test for presence or absence of data [3]. At any point, a process is either enabled or blocked waiting for data. A process cannot wait for data from one queue or another. In Kahn process network, the order in which tokens are produced on the FIFO queues do not depend on the execution order. Systems that follow Kahn's model are determinate. Termination is completely determined by the definition of the process network and does not depend on the execution order.

Computation graphs are a model defined by Karp and Miller, places a restriction on the model that a threshold minimum number of data tokens must be present on the arc before the consumer is fired, which is greater than or equal to the number of tokens consumed. Termination and boundedness are undecidable for process network programs. Any dynamic scheduler for process networks must satisfy two requirements: 1) if a process network program is non-terminating, it must execute forever without terminating, and 2) if possible, the schedule must execute a program so that only a bounded number of tokens ever accumulate on any of the FIFO queues. Parks [1] devised a scheduling policy that simultaneously satisfies both requirements of a dynamic scheduler and provides a desired behavior for all types of programs. The model he proposed for bounded scheduling of process networks has three properties: a) a process is suspended

when attempting to read from an empty queue, b) a process is suspended when attempting to write to a full queue, and c) on artificial deadlock, increase the capacity of the smallest full queue until its producer can fire. Only the read and write operations of Khan process networks program have to be modified to satisfy all the properties mentioned above.

We start the program with an initial estimate on the queue size less than upper bound, such that the program is strictly bounded by the initial bound. We then execute the program satisfying the first requirement using some dynamic scheduling policy. There can be two outcomes: a) the execution of the program stops if and only all the processes are suspended reading from empty queues resulting in a *true deadlock*, or b) the execution may also stop because some processes are suspended attempting to write to full queues resulting in an *artificial deadlock*. If it is true deadlock situation, then the program has terminated. If it is artificial deadlock situation, then we increase the initial bound to a new bound less than the upper bound, and continue execution. This is repeated for a steady forward progress of the process network program. If the program is unbounded, execution repeatedly stops due to artificial deadlock, and we increase the queue size repeatedly without limit. This is the case where we prefer the first requirement to be satisfied and avoid having partial bounded execution.

PRIOR WORK

Computationally intensive sonar beamforming algorithms have been implemented [2] using Process Networks and POSIX Pthreads under Sun Solaris operating system by Greg Allen of ARL at the University of Texas at Austin. This

implementation has been proven to compare favorably with the more traditional thread-pool model, and provides a low-overhead, high-performance, scalable framework. Although the above-mentioned implementation is applied to beamforming, it provides a basic process networks implementation framework [4] that could be used on any appropriate processing task. It uses C++ inheritance mechanism to build interfaces and functionality.

In this implementation, each node of a process networks program corresponds to a POSIX thread and these multiple threads can run concurrently on multiple processors. The computation time of a node is made reasonably larger than the context switch time of a thread. The queues that connect the process nodes are intended to make up for the lack of circular address buffers in general purpose processors, and to prevent unnecessary copying of data. The circular addressing is achieved by mirroring the beginning of the queue's data region(up to some threshold) just past the end of the queue's data region. Thus, queue can provide a pointer to a contiguous block of data elements even when operating near the end of the data region. The queue manages this mirroring, and guarantees that the same data resides in both locations.

This implementation combines the input firing threshold imposed by Computation graph with output firing threshold imposed by Parks bounded scheduling policy. The input and output queue thresholds are dynamic and could be different for different queues and for each transaction on the queues. Similarly, the number of tokens produced and consumed by a node is also dynamic. The

detection of artificial deadlock is necessary only for unbounded programs to execute forever in bounded memory and these programs have no place real-time.

The POSIX condition variable is used to awaken nodes at the proper time. When the producer enqueues data into a queue and there is a suspended consumer waiting at the other end, consumer will be signaled to awaken if the operation provides enough data for the consumer to fire. A similar signaling is done from consumer to producer on dequeueing data.

A general programming model for the applications that is layered on the top of this process networks implementation is as follows:

forever do

get readpointer to a specified block of data to be dequeued

{ blocks until threshold amount of data is available }

get writepointer to a specified block of data to be enqueued

{ blocks until threshold amount of space is available }

copy data contents from readpointer to writepointer

update queue indices based on number of data elements dequeued

{ awaken waiting producer, if enough space available }

update queue indices based on number of data elements enqueued

{ awaken waiting consumer, if enough data available }

done

OBJECTIVES

The main objective of this work is to thoroughly understand the existing bounded memory process networks implementation [2] developed in C++, using

portable Pthread library, and modify the code to handle artificial deadlock. The goal is to modify the existing code without impairing the layered queue implementation approach that is used and to keep it still portable across different platforms. Another goal is to find similar work on the detection and correction of artificial deadlock in bounded scheduling of process networks. However, we could not find related work published in any of the journals as the work on bounded scheduling of process networks is fairly recent.

OTHER RELATED WORK

The process networks domain in Ptolemy II [5] models a system as a network of sequential processes, implemented as Java threads, that communicate using one-way FIFO queues. It implements Parks bounded scheduling and handles detection of artificial deadlock. However, it employs a separate Java thread that handles the deadlocks and resolves them as soon as they arise according to Parks scheduling policies. It differs from our implementation in that every time any thread is suspended on a read or a write, this dedicated Java thread checks for artificial deadlock.

DESIGN AND IMPLEMENTATION

In our design, each queue in the network will have its own size which is fixed at creation time. The user specifies a current queue size and a maximum queue size at the creation time. We allocate queue for maximum size, but manage the queue only with current queue size specified by the user. If one or more threads are blocked writing to a full queue and others blocked reading from an empty queue, we encounter artificial deadlock. Expanding the queue every time

we encounter artificial deadlock is performed by updating the current queue size by a fixed number, adjusting the data elements and performing mirroring of data elements to be consistent with the updated current queue size.

Most of the design changes have been incorporated only in queue management at the process networks layer. In this layer, we maintain a list of *qEntry* classes each containing a queue identifier, queue size and block-type (read or write), sorted by queue size, for all the queues that are created in the network. This list is shared by all the threads in the network. We also maintain some more shared memory information, namely, the number of the threads suspended on read, number of threads suspended on write and the total number of queues at any point in time. All these shared memory information is properly serialized by a *Qmutex* lock for access by all the threads in order to maintain data consistency. The last thread in the network that decides to get suspended (either on read or write) awakens all the threads that are suspended on writing to a full queue. Each awakened thread traverses the *qEntry* list to find if its queue size is the smallest. The thread that has the smallest queue size expands its output queue and attempts to write again. All other awakened threads go back to their suspended state. The thread that detected the artificial deadlock now gets suspended unless it finds itself to be candidate to expand its queue. In both cases, we have either the current thread or the awakened thread running in the system letting the program make forward progress. Thus, we completely eliminated the artificial deadlock situation.

ISSUES AND POSSIBLE ENHANCEMENTS

In order to expand the queue, reallocating the queue every time we encounter artificial deadlock is an option. This is an expensive operation. This might also pose another problem for an application that already has the address of the old queue will fault on further operations on it, once we change to new queue. Our framework must also take care not to acquire locks in the wrong order (lock hierarchy violation) when accessing shared memory variables, as it would deadlock.

We could have a dedicated thread that handles artificial deadlock. Another improvement could be that the last thread that is getting suspended could directly signal the thread that has minimum queue size and waiting on write. Searching the *qEntry* list could be improved.

REFERENCES

[1] T. M. Parks, "Bounded Scheduling of Process Networks," Technical Report UCB/ERL-95-105. EECS Department, University of California, Berkeley, CA 94720, December 1995.

[2] G. Allen, B. Evans, and D. Schanbacher, "Real-time Sonar Beamforming on a Unix Workstation Using Process Networks and POSIX Threads," *Proc. IEEE Asilomar Conference on Signals, Systems & Computers*, pp. 1725-1729, November, 1998.

[3] G. Kahn and D. B. MacQueen, “Coroutines and Networks of Parallel Processes,” *Information Processing*, pages 993-998, Toronto, August 1977.

[4] Process Networks Source code Web Page: http://www.ece.utexas.edu/~allen/PNSourceCode/

[5] Ptolemy II Web Page: http://www.ptolemy.eecs.berkeley.edu/ptolemyII/