Optimizing Parallel Simulation of Multicore Systems Using Domain-Specific Knowledge

Jun Wang, Zhenjiang Dong, Sudhakar Yalamanchili, and George Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA 30332-0250
{jun.wang, zdong30, sudha, riley}@ece.gatech.edu

ABSTRACT

This paper presents two optimization techniques for the basic Null-message algorithm in the context of parallel simulation of multicore computer architectures. Unlike the general, application-independent optimization methods, these are application-specific optimizations that make use of system properties of the simulation application. We demonstrate in two aspects that the domain-specific knowledge offers great potential for optimization. First, it allows us to send Null-messages much less eagerly, thus greatly reducing the amount of Null-messages. Second, the internal state of the simulation application allows us to make conservative forecast of future outgoing events. This leads to the creation of an enhanced synchronization algorithm called Forecast Null-message algorithm, which, by combining the forecast from both sides of a link, can greatly improve the simulation look-ahead. Compared with the basic Null-message algorithm, our optimizations greatly reduce the number of Null-messages and increase simulation performance significantly as a result. On a subset of the PARSEC benchmarks, a maximum speedup of about 6 is achieved with 17 LPs.

Categories and Subject Descriptors

I.6.8 [**SIMULATION AND MODELING**]: Types of Simulation—*Parallel; Discrete event*

General Terms

Algorithms, Performance

Keywords

Null-message algorithm, domain-specific knowledge, optimization, multicore systems, parallel discrete event simulation

1. INTRODUCTION

The current trend in CPU design is moving increasingly toward multicore systems [10], where multiple processing cores are integrated in the same physical chip. Multicore systems bring an extra dimension of complexity, i.e., core-level concurrency and related issues. As always, simulation is playing an important role in helping designers to explore the design space and meet the new challenges. As system complexity increases, researchers have turned to parallel discrete event simulation (PDES) [8] in order to achieve better simulation performance.

The most important difference between PDES and sequential simulation is that events are processed in a distributed manner and the simulation processes, called logical processes (LPs), send messages to each other. To ensure global order of the events, various synchronization algorithms have been created. These are broadly classified as conservative or optimistic algorithms. The *Null-message algorithm*, or the *CMB algorithm* after its inventors [2], is the first synchronization algorithm for PDES. It is a conservative algorithm that maintains correct event order and avoids deadlock by sending the so-called Null-messages. The advantages of the Null-message algorithm are its conceptual simplicity and ease of implementation. However, a commonly faced problem with this algorithm is low performance due to the large amount of Null-messages. Various optimizations of the basic algorithm have been proposed [15][17][6]. Generally speaking, these are general, application-independent optimizations that focus on the algorithmic aspect of the basic scheme. An important fact about the Null-message algorithm is that it depends on a system property called *look-ahead* in order to function. Look-ahead quantifies the simulation process' ability to predict its future and is highly application dependent. Poor look-ahead generally leads to poor performance for the Null-message algorithm. Based on this, we believe that, while general, application-independent optimizations are useful and should be adopted if applicable, optimizations that take advantage of problem-domain knowledge are more effective, particularly if inter-process communication is not heavy. Simulation models of multicore computer systems are just this kind of systems.

This paper presents two optimizations we have created with Manifold [13], our parallel simulation framework for multicore computer architectures. The first optimization differs from the basic scheme in determining when Null-messages are to be sent, and greatly reduces the number of Null-messages and improves performance significantly as a result. In the second optimization, we look inside the data structures of the simulation application and attempt to predetermine the time-stamp of the next output event over an

inter-LP link. Through a simulation kernel API, this information is passed to the synchronization algorithm, which in turn improves its look-ahead when sending out Null-messages. Based on this idea, an enhanced synchronization algorithm called *Forecast Null-message algorithm* is created, where the Null-message carries an LP's forecast of future output events in addition to its time-stamp. By combining the forecast information from both sides of an inter-LP link, the Forecast Null-message algorithm can greatly improve an LP's lookahead. This enhanced algorithm further reduces the number of Null-messages. Although the reduction of the number of Null-messages only produces limited performance improvement when all of the LPs are running on a single physical node, in the more likely and more scalable simulation environment that involves multiple physical nodes, the performance improvement is substantial.

2. BACKGROUND AND RELATED WORK

The Null-message, or CMB, algorithm [2] is the first algorithm created to address the synchronization problem in PDES. The purposes of the algorithm are to avoid deadlocks that may arise because of the conservative principle of event processing, and to insure correct total ordering of events leading to correct event causality. While the algorithm does achieve its goal of guaranteeing causality correctness and preventing deadlocks, it has the potential problem of poor performance due to the excessive number of Null-messages exchanged among the LPs. A number of variations of the basic algorithm have been proposed to address this problem. Two schemes are discussed in [15]. One is to use a time-out such that Null-messages are only sent after a timeout, and another is demand-driven where Null-messages are only sent when requested. In [17], a few variants of a central theme are presented, namely, deferring sending of either Null-messages, or simulation event messages, or both, and combining the messages so as to reduce the messaging overhead. In [6] the author explored ways to reduce the number of Null-messages by predicting the time-stamp of future messages for incoming channels that are current empty. The prediction is based on the underlying network topology, specifically, the feedforward and feedback topologies. These are all optimizations aimed at reducing of the Null-message overhead of the basic algorithm. They are general in nature and independent of the simulation applications. It should be noted that, if the look-ahead of the system can somehow be improved, the number of Null-messages will also be reduced because the simulation time can advance more before an LP needs to send Null-messages. Since look-ahead is heavily dependent on the application, making use of problem-domain knowledge to improve look-ahead can potentially be more effective than the general methods.

The computer architecture community has been using simulation as an effective design and research tool. However, traditionally the simulators being developed and used are sequential. In recent years, a few parallel simulation frameworks have appeared in the computer architecture literature.

SST [16] is an open-source, parallel simulation framework designed for the research of new technologies in the area of supercomputing. It takes a modular approach that allows system models to be built using the components in its repository. For parallel simulation, it uses barriers to address the synchronization problem. For each inter-LP link, a *sync object* is created and is scheduled periodically. When the action associated with the sync object is processed, a global synchronization is performed. Unlike the Null-message algorithm, this is a barrier-based conservative algorithm.

In [4] the authors explore the design space for parallel simulation of chip-multiprocessors and evaluate the impact of a few design choices, including distributed vs. centralized L2 cache, blocking vs. non-blocking cache access, and Null-message vs. barrier-based algorithm. An abstract of the full-fledged simulator is used in the evaluation, and with the Null-message algorithm a speedup of about 2 is achieved with 16 LPs. Apparently the Null-message algorithm used is the basic algorithm without optimizations. We are not aware of any parallel simulators for multicore systems that have explored optimizations of the Null-message algorithm.

Two other parallel simulation systems for computer architectures can be classified as optimistic simulation because they allow violations of the local causality constraint. In other words, they allow external events to arrive in an LP's past. *SlackSim* [3] defines a quantity called *slack* which basically specifies how far in the simulation time an LP can go from its current local time. A separate thread monitors the smallest local time of all LPs, i.e, the global virtual time (GVT). As the GVT moves, the upper bound for event execution also moves. This makes it different from quantum simulation, which uses explicit barrier synchronization. *Graphite* [14] is similar to SlackSim in that it allows causality violations. It supports three simulation schemes: *Lax*, *Lax-Barrier*, and *Lax-P2P*. *Lax* is essentially unlimited optimism. *Lax-Barrier* is like a quantum simulation, and in *Lax-P2P* an LP periodically chooses a random peer with which to synchronize. These methods are fundamentally different from the Null-message algorithm, which executes events in a conservative manner and does not allow simulation errors.

3. THE MANIFOLD SIMULATION SYSTEM

Manifold [13] is a scalable parallel discrete event simulation framework designed for the research of modern multicore computer architectures. Manifold adopts a component-based open software architecture, which, coupled with standardized inter-component interfaces, allows easy integration of third party components. The software package is composed of a parallel simulation kernel and a set of computer architecture models, such as processor, cache, interconnection network, and DRAM controller models. Users use the kernel and the models to build system models for parallel simulations. A typical architecture that Manifold is designed to simulate is the tiled multicore system as discussed in [10]. This architecture is made up of a number of identical processor nodes connected with an interconnection network. The system model is shown in Figure 1. As can be seen, the system model consists of a few processor nodes and one or more memory controller nodes, interconnected through the underlying interconnection network. The processor nodes each consist of a processor core and one or more level of cache. The dashed lines in Figure 1 shows the partitioning scheme used by our simulator, which will be explained later.

Manifold adopts a layered software architecture [5], sepa-

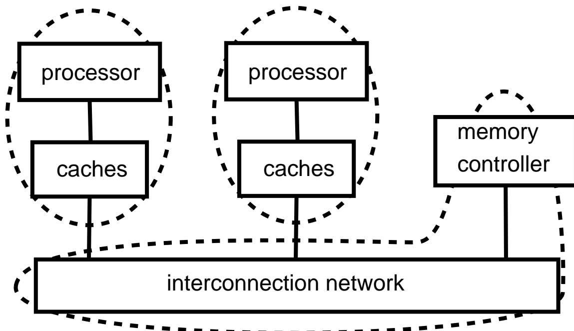

Figure 1: Manifold's example system model and partitioning scheme.

rating the simulation kernel and the computer architecture components into two different layers. The simulation kernel layer encapsulates all the PDES services. For parallel simulation, two conservative synchronization algorithms are supported, Null-message and lower bound time-stamp (LBTS) [8]. Quantum simulation is also supported, for users who can allow a small percentage of inaccuracy in exchange for faster simulation speed.

4. OPTIMIZING THE NULL-MESSAGE ALGORITHM IN MANIFOLD

Manifold does not use automatic partitioning, mainly because the number of components in a system model is relatively small. Instead, components are assigned to different LPs in the simulator program. A typical partitioning scheme we have been using is shown in Figure 1 with dashed lines denoting LP boundaries. As can be seen, in this partitioning scheme, the entire interconnection network and the memory controller(s) are in one LP, and each processor node is in a separate LP. In the following we shall assume this partitioning choice.

Another important characteristics of Manifold that should be mentioned in advance is that the system model represents a clock based digital system. Although Manifold allows link delays to be specified in seconds, in this paper we confine our discussion to the cases where link delays are specified in clock ticks. Therefore, all events occur at discrete clock ticks, and the simulation moves from one clock tick to the next.

4.1 Baseline algorithm

We start with a basic implementation of the Null-message algorithm, as shown in Algorithm 1. This is the Null-message algorithm in its most general form. It is application-independent, therefore, does not make use of any characteristics of the simulation application.

In this baseline algorithm, the LP first receives messages in a non-blocking fashion. It then finds out the clock whose next edge is the earliest (There may be multiple clocks). The simulation time of this particular clock edge is kept in the variable `nextClockTime`. Next, the minimum time-stamp of Null-messages from all of the input channels is determined and kept in `min_null`. If `nextClockTime` is less than `min_null`, then all of the events scheduled for the clock edge are safe to process and therefore are processed. Finally, Null-messages are sent to all successors of the LP with time-stamp set to the smaller of `nextClockTime` and `min_null`, plus the look-ahead value. Of course, Null-messages are only sent if the time-stamp is larger than that of the last Null-messages. The look-ahead is set to a value that is slightly smaller than the minimum link delay in the system model.

Algorithm 1 Baseline Null-message algorithm.

1: **while** simulation not terminated **do**
2: receive messages
3: $nextClock$:= clock whose next edge has smallest time-stamp among all clocks
4: $nextClockTime$:= simulation time of next edge of $nextClock$
5: receive Null-messages
6: min_null := minimum time-stamp of Null-msgs from all input links
7: **if** $nextClockTime < min_null$ **then**
8: process all events scheduled for the clock edge
9: **end if**
10: $null_ts := min(nextClockTime, min_null) + Lookahead$
11: Send Null-messages to all successors with time-stamp set to $null_ts$ if $null_ts$ is larger than time-stamp of the previous Null-message
12: **end while**

4.2 Optimization 1: When To Send Null-Messages

The baseline algorithm in Algorithm 1 does achieve its basic design goals, namely, to avoid deadlock and to prevent causality errors. Its drawback is that a lot of Null-messages are sent, most likely more than necessary. This has a big impact on the overall simulation performance, particularly when the messages have to go through a network.

Therefore, we start our optimization of the baseline algorithms by exploring ways to reduce the sending of Null-messages. Two optimization algorithms have been tested with success. It should be pointed out that both these algorithms make use of features of the simulation application, i.e., the multicore computer architecture. Therefore, these are application-specific optimizations.

The first algorithm shall be referred to as Send-When-Safe, or *SWS*. In this algorithm, shown in Algorithm 2, Null-messages are only sent when the clock edge is safe to process.

Algorithm 2 SWS: send Null-message when clock edge is safe.

1: **while** simulation not terminated **do**
2: receive messages, determine nextClock and nextClockTime, receive Null-messages, determine min_null
3: **if** $nextClockTime < min_null$ **then**
4: process all events scheduled for the clock edge
5: send Null-message with time-stamp set to $nextClockTime + Lookahead$
6: **end if**
7: **end while**

SWS is possible because of the following properties of the simulation application:

System Property 1. *Each LP moves in the fixed step of half clock tick.*

System Property 2. *The look-ahead is bigger than half clock tick.*

To see the simulation can progress without deadlock, we can simply look at one LP. Assume the minimum link delay in the system is 1 tick and the look-ahead is set to 0.9 ticks, and denote by $Null(ts)$ the Null-message with time-stamp set to ts. First, the LP processes tick 0, which is safe, and sends out $Null(0.9)$. Next, tick 0.5 cannot be processed until the minimum time-stamp of all the Null-messages it gets from the input links is greater than 0.5. This is guaranteed because, with the properties above, all of its fan-ins would process tick 0 and send $Null(0.9)$ to it. When that occurs, it processes tick 0.5 and sends $Null(1.4)$. Then it moves on to tick 1. Eventually it will receive $Null(1.4)$ from all its fan-ins and be able to process tick 1, and so forth.

The second optimization algorithm, Send-When-Block, or *SWB*, does the opposite of *SWS*. It sends Null-messages only when it is blocked, as shown in Algorithm 3.

Algorithm 3 SWB: send Null-message when clock edge is blocked.

```
1: while simulation not terminated do
2:     receive messages, determine nextClock and nextClock-
       Time, receive Null-messages, determine min_null
3:     if nextClockTime < min_null then
4:         process all events scheduled for the clock edge
5:     else
6:         send Null-message with time-stamp set to min_null +
           Lookahead
7:     end if
8: end while
```

It can be easily verified that this algorithm is correct in general, independent of the simulation application. However, this algorithm forces a certain degree of sequential execution. For any LP, once it is blocked, it can only make progress after all of its fan-ins have made progress, and its fan-outs can only make progress after it has made progress.

This, however, is not a serious problem for our simulation model. With the star topology shown in Figure 1. the *SWB* algorithm causes the simulation to progress in lock-step between two groups of LPs. On the one hand is the center of the star, or the network+memory controller, and on the other hand is the rest of the system, i.e,. the processor nodes. Since the processor nodes are much more complex and perform most of the simulation work, and they still do the work in parallel, this lock-step execution between the two groups has only limited impact. Of course, if the workload of the star center is much heavier, it may be necessary to send more Null-messages if parallelism can be improved.

4.3 Optimization 2: Improving Look-ahead

Look-ahead plays a critical role in the Null-message algorithm. Generally speaking, the bigger the look-ahead, the better the parallelism, and hence the better the performance. Therefore, our next optimization focuses on improving look-ahead in our simulation.

As mentioned above, we set our look-ahead to a value that is close to the minimum link delay in the system. To be precise, it is set to $(d_{min} - 0.1)$ ticks, where d_{min} is the minimum link delay. This can be improved if we look a little deeper into the LPs. In the following, we explain how we improve the look-ahead for *Iris*, our interconnection network model, and *MCP-cache*, our coherence cache model.

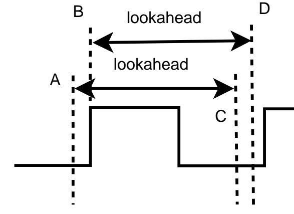

Figure 2: Look-ahead improvement due to synchronous output.

Iris consists of a number of routers connected in a ring or torus topology. Each router is connected to one network interface (NI), and vice versa. An NI is connected to a single *terminal*, such as a cache component or a memory controller.

In the partitioning scheme in Figure 1 the link between the network interface and MCP-cache crosses LP boundary. We can make two improvements for the look-ahead for this link.

First, we observe that a component in our system models sends out messages at the rising clock edges. This is a system-wide property and applies to both Iris and MCP-cache.

System Property 3. *A component only sends out messages at the rising clock edges.*

This means that the time-stamp for the Null-messages in line 6 in *SWB* in Algorithm 3 can be set to $\lceil t \rceil + Lookahead$ instead of $(min_null + Lookahead)$, where the meaning of $\lceil t \rceil$ is:

$$\lceil t \rceil = \begin{cases} \text{current edge} & \text{if t is rising edge} \\ \text{next rising edge} & \text{if t is falling edge} \end{cases}$$

This is further illustrated in Figure 2 which shows four points in time: A is *min_null*, or the minimum Null-message time-stamp from all input links, B is the current edge for which the safety of processing is being determined, $C = A + Lookahead$ is the time-stamp of the Null-messages to be sent, based on *SWB* in Algorithm 3, and D is the improved time-stamp for the Null-messages based on the fact that the earliest event to be sent will be sent at the clock edge B. If B is a falling edge, then the Null-message time-stamp would be set to the next rising edge plus the look-ahead.

Note that this improvement cannot be made in the general Null-messages-based simulation, because an event could occur between A and B and cause another event that is earlier than B. So, in the general case, when B is not safe to process, the time-stamp of the Null-messages can only be $A + Lookahead$.

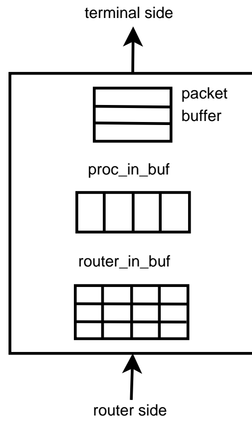

Figure 3: Iris network interface.

This improvement is shown in Algorithm 4.

Algorithm 4 Improvement of SWB: using next rising edge.

```
1: while simulation not terminated do
2:     receive messages, determine nextClock and nextClock-
       Time, receive Null-messages, determine min_null
3:     if nextClockTime < min_null then
4:         process all events scheduled for the clock edge
5:     else
6:         if nextClockTime is rising edge then
7:             null_ts := nextClockTime + Lookahead
8:         else
9:             null_ts := next rising edge +Lookahead
10:        end if
11:        send Null-message with time-stamp set to null_ts
12:    end if
13: end while
```

The second improvement of look-ahead makes use of the knowledge of the internal state of Iris and MCP-cache. Based on the internal state, we can forecast *conservatively* the earliest time when Iris or MCP-cache will send out a message and use this information to improve look-ahead. In the following, we first describe how the conservative forecast is achieved in Iris and MCP-cache without considering flow control messages. Then we describe the complications caused by flow control, and how the forecast is made in the presence of flow control messages. And finally we present the improved algorithm that uses the forecast.

Iris' routers operate at the flit level [7]. When a packet is received for delivery, at the source NI, the packet is broken up into flits and passed to the router one flit at a time. For each packet, there is one head flit, zero or more body flits, and zero or one tail flits. At the receiving NI, the flits for the packet are assembled. When the entire packet has been received, it is put in the output buffer before being delivered to the terminal. Figure 3 shows the internal structure of the Iris NI for the router-to-terminal direction of traffic. The other direction has a similar structure. As can be seen, there are three buffers. The *router_in_buf* holds individual flits for each of the virtual channels. The *proc_in_buf* is where the flits are assembled into packets. It has one slot for each virtual channel. Finally the *packet buffer* holds the assembled packets before delivery.

The system properties of Iris that we use to forecast its output messages are summarized as follows:

System Property 4. *If the packet buffer of an Iris NI is empty, then the earliest time it will send out a packet is the next tick.*

System Property 5. *If the last flit of a packet arrives at the NI's proc_in_buf at tick t, then the earliest time when the packet is moved to the packet buffer is $t + 1$.*

System Property 6. *The router sends at most one flit per tick to the NI.*

System Property 7. *The router has a 3-stage pipeline. Therefore, it takes at least three ticks for a head flit to go through the router.*

Using these properties, Iris NI can make conservative forecast of the earliest possible tick when it will send out a message to its terminal.

On the MCP-cache side it is much simpler. The cache has a parameter called *lookup time* that represents the number of ticks it takes for the cache to send out a message as a result of an incoming request. Furthermore, we can know L ticks in advance if there is an outgoing message, where L is the lookup time. This is summarized below.

System Property 8. *When a request is received by the MCP-cache at tick t, the earliest time when a message is sent out as a result of the request is $t + L$, where L is the lookup time. And if there is a message, it is known at t.*

Although the above system properties of Iris and MCP-cache allow us to forecast outgoing messages, this information cannot be used right away to improve the look-ahead. This is because both Iris and MCP-cache implement a credit-based flow control mechanism. In addition to normal event messages, both Iris and MCP-cache send credit messages, which could violate our forecast of outgoing event messages. The credit message has the following property.

System Property 9. *For each received event message, a credit message is sent.*

The difficulty lies in predicting when a credit is sent, which is component dependent. MCP-cache sends a credit one tick after a message is received, while Iris sends a credit only after a packet is moved from the terminal-side input packet buffer to the router-bound middle buffer. For simplicity we assume the worst case, namely, a credit can be sent any time after a message is received.

Algorithms 5 and 6 show the outgoing message forecast algorithm for Iris and MCP-cache respectively. Both functions

Algorithm 5 Iris' algorithm for forecasting outgoing message.

```
 1: function IRIS_FORECAST_NEXT_OUTPUT
 2:     if outgoing credit scheduled for this tick then
 3:         return 0                                   ▷ this tick
 4:     if output packet buffer not empty then
 5:         return 0                                   ▷ this tick
 6:     if an output packet will be assembled this tick then
 7:         return 0                                   ▷ this tick
 8:     if there is an input message then
 9:         m_ts := input message scheduled time
10:         if (m_ts < now) then
11:             return 0              ▷ credit could be sent any time
12:         else
13:             forecast := m_ts − now
14:         end if
15:     else                      ▷ no un-credited incoming messages
16:         for each virtual channel v do
17:             if (proc_in_buf[v] has flits) then
18:                 forecasts[v] := packet len - received flits
19:             else                       ▷ proc_in_buf[v] is empty
20:                 if (router_in_buf[v] has flits) then
21:                     forecasts[v] := packet len
22:                 else                   ▷ router_in_buf[v] empty
23:                     forecasts[v] := router → earliest() + 2
24:                 end if
25:             end if
26:         end for
27:     end if
28:     forecast := min(forecasts[])
29:     return now + forecast
30: end function
31: function ROUTER::EARLIEST
32:     if there is one in-transit packet bound for NI then
33:         return 0
34:     else
35:         return 3
36:     end if
37: end function
```

Algorithm 6 MCP's algorithm for forecasting outgoing message.

```
 1: function MCP_FORECAST_NEXT_OUTPUT
 2:     if L1's output buffer not empty OR L2's output buffer not
    empty then
 3:         return 0                                   ▷ this tick
 4:     forecast := 0
 5:     if there is msg or credit scheduled then
 6:         forecast := earliest msg or credit time-stamp
 7:     if (forecast == now) then
 8:         return 0                                   ▷ this tick
 9:     if no msg or credit scheduled then
10:         forecast := min(L1's lookup time, L2's lookup time)
11:     return now + forecast
12: end function
```

return 0 if an outgoing message could occur at the current tick, otherwise, they return the earliest possible tick for the next outgoing message.

In Algorithm 5, lines 2-7 are self-explanatory. Lines 9-14 deal with the problem of making forecast in the presence of un-credited incoming messages, which require credits to be sent. To handle outstanding credits, we use a FIFO queue. Every time there is an incoming message, its scheduled time is entered into the queue. When a credit is sent, the first element in the queue is removed. When there is an un-credited incoming message scheduled at *m_ts*, if its already past, then the credit could be sent at any moment, therefore, we return 0. Otherwise, since the credit for the message could be sent at *m_ts*, we make the conservative forecast that the earliest possible time for an outgoing message is *m_ts* (line 13).

Lines 16-26 handle the case where there are no outstanding credits to be sent. In this situation, we don't have to worry about sending out credits. Therefore we look at the NI's buffers to forecast the next outgoing event message. For each virtual channel, we first look at *proc_in_buf*, where the packet is assembled. If *proc_in_buf* is not empty (line 18), since the head flit carries the packet length in terms of the number of flits, we know how many more flits are yet to come, and since we can only receive one flit per tick, that is the lower bound of the remaining ticks it takes before we have a complete packet. If *proc_in_buf* is empty, we check the *router_in_buf*. If it is not empty (line 21), then it must holds the head flit because *proc_in_buf* is empty. Therefore, it takes at least $packetlen - 1 + 1$ before all the flits for the packet enters *proc_in_buf*. If *router_in_buf* is also empty (line 23), then we consult the router for the earliest possible time when it could forward a head flit to the NI on the given virtual channel. We add two ticks to the router's forecast (one for entering *router_in_buf*, and one for entering *proc_in_buf*). Finally, in line 28, we find out the minimum forecast value from all of the virtual channels and use that as our forecast for the Iris NI.

The forecast algorithm for MCP-cache is much simpler because: (1) its internal structure is simpler, and (2) it always sends out a credit one tick after receiving a message. We use a priority queue to hold the ticks of future outgoing event messages and credits. If this queue is not empty, the first element is used as the forecast. Otherwise, the smaller of L1 and L2 caches' lookup time is used as the forecast.

We now present the improved Null-message algorithm that uses the event forecast as described above to improve lookahead. For this purpose the Null message itself is slightly enhanced such that it carries a pair $Null(ts, forecast)$, where the meaning of the time-stamp ts is as usual, and $forecast$ means the sender will not send a message before $forecast$ unless it receives a message between ts and $forecast$. The enhanced Null-message is shown in Figure 4. The basic idea of the algorithm is to use the minimum of the forecast values from both sides of a link as the basis for the time-stamp of the next Null-message.

The Forecast Null-message algorithm is shown in Algorithm 7.

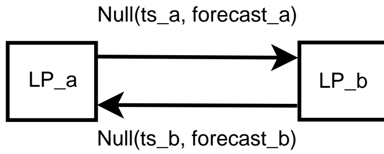

Figure 4: Null-message with event forecast.

Algorithm 7 The Forecast Null-message algorithm.

```
1: while simulation not terminated do
2:     receive messages, determine nextClock and nextClock-
       Time, receive Null-messages, determine min_null
3:     if nextClockTime < min_null then
4:         process all events scheduled for the clock edge
5:     else
6:         for each registered component c do
7:             c → forecast_next_output() to determine the next
               possible cross-LP output event
8:         end for
9:         min_null_ts := 0
10:        for each successor do
11:            out_forecast := time of next possible output
12:            in_forecast := forecast of next possible input from
               successor
13:            if out_forecast > in_forecast then
14:                forecast := in_forecast + 1          ▷ assuming
15:                                                ▷ LP-to-LP delay is 1
16:            else
17:                forecast := out_forecast
18:            end if
19:            null_ts := nextClockTime + Lookahead
20:            if forecast > nextClockTime then
21:                null_ts := forecast + Lookahead
22:            end if
23:            if (min_null_ts==0 || min_null_ts>null_ts) then
24:                min_null_ts := null_ts
25:            end if
26:        end for

27:        for each successor do
28:            send Null-message with time-stamp set to
               min_null_ts
29:        end for
30:    end if
31: end while
```

As can be seen, Algorithm 7 is a modification of *SWB*. It also sends Null-messages only when it is blocked.

Lines 6-8 are where the forecast is made. Each component that is interested in making forecast should register its forecast function with the clock object. At a rising edge of the clock when the LP's execution is blocked, the registered functions are called.

In the loop in lines 10-26 we find out the time-stamp for the next Null message for each successor LP. The forecast is used as follows. The LP sending the Null-messages will compare its forecast for the successor ($out_forecast$) and the successor's forecast carried in the received Null-message ($in_forecast$). If $out_forecast > in_forecast$, then we can guarantee we will not send out a message before $in_forecast+1$, assuming the minimum inter-LP delay is 1. Otherwise, the earliest time when we are going to send a message is simply $out_forecast$.

We find out the minimum time-stamp for the Null-message for all of the successors, and then in the loop in lines 27-29, we send out the Null-messages. Using the minimum is because experiments have shown this gives better performance results.

4.4 Section Summary

This section has presented two optimizations for the Null-message algorithm, particularly, an enhanced Null-message algorithm called Forecast Null-message algorithm. A few important points should be noted regarding the applicability of the methods.

- The optimization methods use domain-specific knowledge. It should be pointed out that the knowledge used is generally available in the application domain, namely, multicore system models. For example, a simulator for multicore systems is generally clock-cycle-based, a cache model usually has an access latency, and a router of an interconnection network is generally pipelined. Therefore, the methods are by no means limited to the particular components or simulation system that we used and are applicable to other similar systems.
- The methods are not dependent on the partitioning scheme shown in Figure 1. This partitioning scheme is based on the observation that, because cache hit rate is generally well over 95%, only a small amount of messages are exchanged between the caches and the network. If, for example, a different way of partitioning puts two neighboring routers in two different LPs, the optimizations presented are still applicable, except that we need to look inside the routers to in order to use the Forecast Null-message algorithm.
- The domain-specific optimizations do not preclude the general optimization methods such as the idea of deferring the sending of Null-messages as presented in [17]. However, how well the two kinds of optimizations work together in a single simulation system is yet to be studied.

5. EXPERIMENTAL RESULTS

We have implemented the four algorithms described above: *Baseline*, *SWS*, *SWB*, and *Forecast*. This section describes the experimental results obtained with the four algorithms, and presents our analysis of the results.

5.1 Experiment Environment

Tests are conducted on a standalone machine and on a Linux cluster. The standalone machine has two Intel Xeon E5-2620 6-core CPUs with hyperthreading, for a total of 24 hardware threads. The operating system is Debian Wheezy and the MPI package used is OpenMPI 1.4.3. The two nodes that we use on the cluster each have two Intel Xeon X5670 6-core CPUs with a total of 24 hardware threads. The operating system is RHEL release 6.3, and the MPI package is OpenMPI 1.5.4.

We randomly selected five programs from the PARSEC 2.0 benchmark suite [1] for our testing. These are widely used

parallel benchmark programs. A program that uses Intel's Pin API [9] was used to generate trace files from the binaries of the benchmark programs. The trace files are fed to the simulator as input.

The simulation system model consists of 16 processor nodes, a memory controller node, and a torus interconnection network, like the model shown in Figure 1. Each processor node has a processor component, a private L1 cache, and a shared L2 cache slice. The processor component is a cycle-accurate out-of-order x86 model called Zesto [12]. The caches implement a directory-based MESI (Modified-Exclusive-Shared-Invalid) coherence protocol to maintain cache coherence. The lookup time is set to 5 ticks for both L1 and L2 caches. For simplicity, there is one clock used by all of the components.

For each tested benchmark, 16 trace files are used as input, one for each processor component. A multicore shared-memory machine emulator called QSim [11] was used in generating the trace files.

As shown in Figure 1 the network and the memory controller are assigned to one LP, and the processor nodes are each assigned to a separate LP, for a total of 17 LPs.

5.2 Results

Two sets of tests have been conducted. In the first set, all of the LPs are allocated to the same machine. And in the second set, the LPs are allocated to two machines.

5.2.1 Test Set 1

This Test Set is run on the standalone machine. Results for the Test Set are given in Tables 1 - 3 and Figure 5. We run the simulation for 10 million simulated clock cycles. Each data item in the table is the average value of three runs.

Table 1 shows the number of Null-messages sent/received by the interconnection network component, Iris, per link. Each entry has two numbers, for sent and received Null-messages respectively. With the baseline algorithm, a large number of Null-messages are exchanged. In fact, the average interval between Null-messages is less than 0.2 ticks, even though the look-ahead is 0.9 ticks. The number of Null-messages in *SWS* can be predicted. It sends two every tick per link. The interval between two Null-messages is half a tick. Compared with the baseline algorithm, this is a big improvement. *SWB* reduces the number of Null-messages even further. The lock-step execution between the network component and the processor nodes results in the fixed interval of 1.8 ticks between two successive Null-messages. This is twice the look-ahead. Finally, *Forecast*, by dynamically improving the look-ahead of both sides of the cross-LP links, achieves an additional 29 to 61% reduction of the number of Null-messages compared with *SWB*. On the Iris side, the Null-message frequency is reduced to one in 3.5 to 3.9 ticks, with the exception of *bodytrack*. On the cache side, the frequency is one in 3.9 to 5.8 ticks.

Table 2 shows the simulation running time of the tests. Table 3 shows the performance improvements of the Forecast Null-message algorithm over SWB, and the speedup numbers relative to the sequential simulation. The table shows a maximum speedup of 6.3. In Figure 5 we show the running time of the three optimization algorithms, normalized with respect to the running time of SWB.

Table 1: Number of Null-messages per link.

Benchmarks	Baseline	SWS	SWB	Forecast
bodytrack	81,462,560	20M	5,555,559	5,298,071
	78,362,448	20M	5,555,560	2,554,396
facesim	73,753,704	20M	5,555,557	2,832,306
	66,864,855	20M	5,555,557	1,748,367
freqmine	67,549,584	20M	5,555,557	2,833,459
	60,451,605	20M	5,555,557	1,760,523
streamcluster	68,659,786	20M	5,555,557	2,558,872
	61,198,045	20M	5,555,557	1,704,793
vips	82,331,880	20M	5,555,557	2,571,159
	75,666,786	20M	5,555,557	1,715,470

As we can see, compared with the baseline algorithm, all of *SWS*, *SWB*, and *Forecast* reduce the simulation time by more than 60%. However, among the three optimization algorithms, the situation looks more complex. For example, compared with *SWS*, *SWB* has significantly fewer Null-messages, but this does not translate into reduction in running time. On the contrary, *SWS* outperforms *SWB* in 4 cases. On the other hand, *Forecast* outperforms *SWB* by 8-12%, while the Null-message reduction is much bigger, in the range of 29 - 61% In the case of *freqmine*, *Forecast* only outperforms *SWS* by 2.3% while we have an 88% reduction in the amount of Null-messages. In this test setting, all of the LPs are running on the same shared-memory machine. The overhead of inter-process messaging is very low to start with. The results suggest two conclusions. First, in this environment, reducing the number of Null-messages has diminishing returns. Second, the results show *SWS* outperforms *SWB* in 4 cases even though its Null-message count is 3-4 times higher. This indicates some other factor may play a more important role than the number of Null-messages. One possibility is that, because *SWS* sends Null-messages more frequently than *SWB*, it may have better run-time parallelism. We leave to our future work the study of the tradeoff between the number of Null-messages and parallelism.

Table 2: Simulation running time in seconds.

Benchmarks	Seq.	Baseline	SWS	SWB	Forecast
bodytrack	2410	2592	856	868	769
facesim	4323	2154	869	825	722
freqmine	4457	2099	764	814	746
streamcluster	4630	1957	761	815	733
vips	4508	2446	769	807	719

Table 3: Forecast Null-message: improvements over SWB and speedup.

Benchmarks	Improvement	Speedup
bodytrack	11.4%	3.1
facesim	12.5%	6.0
freqmine	8.4%	6.0
streamcluster	10.1%	6.3
vips	10.9%	6.3

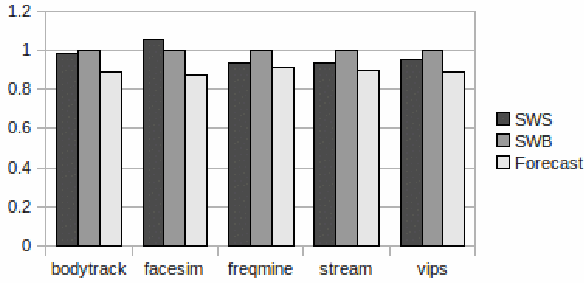

Figure 5: Test Set 1 normalized running time.

5.2.2 *Test Set 2*

This Test Set is run on the cluster. In this Test Set, we allocate the 17 LPs roughly evenly to the two test machines, with 9 LPs on one machine and 8 on another. This makes half of inter-LP links cross the machine boundary, therefore, messages on these links, including Null-messages, will go through the physical network. We expect the number of Null-messages has a much bigger impact on the simulation performance than in Test Set 1.

Results are shown in Tables 4 - 6, and Figure 6. We run the simulation for 10 million simulated clock cycles. Each data item in the tables is the average value of three runs. Results for the baseline Null-message algorithm are not collected simply because it takes too long (over 24 hours) to finish one run, making it an unviable choice in this testing environment.

Table 4: Number of Null-messages per link.

Benchmarks	SWS	SWB	Forecast
bodytrack	20M	5,555,559	5,824,030
	20M	5,555,558	2,622,173
facesim	20M	5,555,559	2,737,507
	20M	5,555,558	1,730,305
freqmine	20M	5,555,989	2,841,734
	20M	5,555,989	1,736,944
streamcluster	20M	5,555,558	2,553,496
	20M	5,555,558	1,679,566
vips	20M	5,555,560	2,586,544
	20M	5,555,559	1,669,547

Table 5: Simulation running time in seconds.

Benchmarks	Seq.	SWS	SWB	Forecast
bodytrack	2326	1915	1304	934
facesim	4340	1658	1178	1006
freqmine	4210	2091	1321	913
streamcluster	4080	1963	1297	811
vips	4023	1592	1179	723

Table 4 shows the number of Null-messages sent/received by Iris, Table 5 shows the simulation running time, Table 6 shows the performance improvements of the Forecast Null-message algorithm over SWB, and its speedup numbers relative to the sequential simulation, and Figure 6 shows the running time normalized with respect to SWB. Compared with Test Set 1, it seems the number of Null-messages has a much bigger impact on the performance. For example, in Test Set 1, the running times of *SWS* and *SWB* are close, with *SWS* outperforms *SWB* in a few cases. Here we see *SWB* outperforms *SWS* consistently by a significant amount: 26-36%. Comparing *Forecast* with *SWB*, we see a reduction in simulation time of 14-38%. Therefore, we can state that reducing the number of Null-messages is very important in the case where inter-LP links cross physical machine boundaries.

Table 6: Forecast Null-message: improvements over SWB and speedup.

Benchmarks	Improvement	Speedup
bodytrack	28.4%	2.5
facesim	14.6%	4.3
freqmine	30.9%	4.6
streamcluster	37.5%	5.0
vips	38.7%	5.6

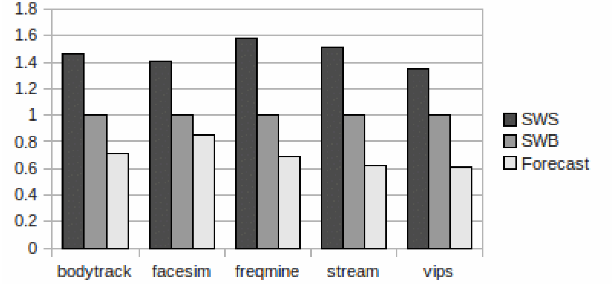

Figure 6: Test Set 2 normalized running time.

6. CONCLUSIONS AND FUTURE WORK

Two optimizations of the Null-message algorithm using domain-specific knowledge are presented in the context of parallel simulation of multicore systems. Because of the characteristics of the simulation application, we can send Null-messages much less eagerly than the baseline algorithm. In addition, the internal state of the LP together with the partitioning topology allows us to make conservative forecast of future events. Based on this, an enhanced algorithm called Forecast Null-message algorithm is created, in which the Null-message is enhanced to carry the forecast information. And with the forecast from both sides of the inter-LP link we can achieve a dynamic look-ahead that is much greater than what can be obtained statically from the inter-LP delays. The optimizations greatly reduce the number of Null-messages compared with the baseline algorithm. When the simulation is run on multiple physical machines, the reduced number of Null-messages has a great impact on the performance. However, if all the LPs run on the same machine, the reduction of Null-messages has diminishing returns. For our future work, we intend to investigate the scalability of the methods, the impact of different partitioning schemes, and other possible factors that could play an important role in improving simulation performance.

7. REFERENCES

[1] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. *Proceedings of the 5th*

Annual Workshop on Modeling, Benchmarking and Simulation, 2009.

[2] K. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, 1979.

[3] J. Chen, L. Dabbiru, D. Wong, M. Annavaram, and M. Dubois. Adaptive and speculative slack simulations of cmps on cmps. *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 523–534, 2010.

[4] M. Chidester and A. George. Parallel simulation of chip-multiprocessor architectures. *ACM Transactions on Modeling and Computer Simulation*, 12(3):176–200, July 2002.

[5] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2nd edition, 2011.

[6] R. DeVries. Reducing null messages in misra's distributed discrete event simulation method. *IEEE Transactions on Software Engineering*, 16(1):82–91, January 1990.

[7] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks, an Engineering Approach*. Morgan Kaufmann, 2003.

[8] R. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, 2000.

[9] Intel. Pin - a dynamic binary instrumentation tool. `http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[10] S. Keckler, K. Olukotun, and H. Hofstee, editors. *Multicore Processors and Systems*. Springer, 2009.

[11] C. Kersey, A. Rodrigues, and S. Yalamanchili. A universal parallel front-end for execution driven microarchitecture simulation. *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation Methods and Tools*, pages 25–32, 2012.

[12] G. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. *International Symposium on Performance Analysis of Software and Systems*, pages 53–64, 2009.

[13] manifold.gatech.edu. Manifold. `http://manifold.gatech.edu`.

[14] J. Miller, H. Kasture, G.Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. *Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[15] J. Misra. Distributed discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[16] A. Rodrigues, K. Hemmert, B. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B.Jacob. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, March 2011.

[17] W.-K. Su and C. Seitz. Variants of the chandy-misra-bryant distributed discrete-event simulation algorithm. Technical Report Caltech-CS-TR-88-22, California Institute of Technology, 1988.