FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. (02461) 61-6402

Technical Report

CESRI 2007 Research Report
Implementation and Validation of the Extended Controlled Logical Clock

John C. Linford

FZJ-JSC-IB-2007-11

November 2007
(last change: 02.11.2007)

Contents

List of Figures

List of Tables

Chapter 1

Abstract

CESRI is a fellowship opportunity sponsored by the National Science Foundation and managed by the Institute of International Education for U.S. graduate students in science and engineering who are seeking a quality hands-on international research experience in Austria, the Czech Republic, Germany, Hungary, Poland, and Slovakia. The Extended Controlled Logical Clock is a method for correcting invalid timestamps in event trace files during post-mortem performance analysis. Under the CESRI 2007 program, I developed a highly-scalable implementation of the Extended Controlled Logical Clock for use in the SCALASCA performance analysis toolkit and verified the method's applicability to real-world supercomputing applications. This document describes the theory of the Extended Controlled Logical Clock and the design of a highly-scalable implemenation of the algorithm. It also gives a narative description of the professional impact of the CESRI fellowship, and the cultural and personal experiences which made my time as a 2007 CESRI fellow exceptional.

Chapter 2

Project Proposal and Overview

CESRI is a fellowship opportunity sponsored by the National Science Foundation and managed by the Institute of International Education for U.S. graduate students in science and engineering who are seeking a quality hands-on international research experience in Austria, the Czech Republic, Germany, Hungary, Poland, and Slovakia. CESRI hopes to improve the expertise of the awardees as scientists, help them think and analyze on a broader, global level, build individual and institutional partnerships, and build dialogue between the scientific community in the U.S. and Central Europe. My research proposal for the CESRI 2007 program was to "alleviate scalability, portability and usability problems in high-performance computing software systems." I achieved this by improving the SCalable performance Analysis of LArge SCale Applications (SCALASCA) toolkit, which performs post-mortem analysis of message-passing applications. I implemented the Extended Controlled Logical Clock as part of SCALASCA and instrumented two multiphysics air quality models with SCALASCA, demonstrating its applicability to massively-scalable, high-performance applications.

The purpose of this document is twofold. First, as a report on my experiences as a CESRI 2007 fellow, it discusses the professional and academic gains which came through the CESRI program and the personal experiences I had as an American researcher abroad. Second, as a technical report, this document gives a detailed explanation of the Extended Controlled Logical Clock (ECLC) in terms of the SCALASCA event model, and describes a highly-scalable implementation of the ECLC for use in SCALASCA.

The document is organized as follows. An introduction to the Extended Controlled Logical Clock is given in Chapter 3, and a description of the implementation is given in Chapter 4. The narative description of my personal and professional experience as a CESRI fellow is found in Chapter **??** and Chapter **??**.

Chapter 3

The Extended Controlled Logical Clock

Post-mortem analysis of message-passing applications provides measurements of wait-times. An analysis of these wait-times can detect system errors and program design flaws in large-scale supercomputing applications, but the accuracy of the analysis depends on the comparability of event timestamps taken on different processors. An inaccurate timestamp will not only dialate the interval between events, but may alter the logical order of events, causing a message to appear to be received before it is sent. This is a violation of the *clock condition* which requires that when an event e with timestamp $C(e)$ precedes an event e' with timestamp $C(e')$, then $C(e) < C(e')$.

Unfortunately, processor clocks are often entirely non-synchronized or only synchronized in disjoint partitions (i.e., an SMP-node or multicore-chip). Clock synchronization protocols, such as NTP [4], are typically too inaccurate for our purposes, but assuming that all local clocks on a parallel machine run at different but constant speeds (i.e., drifts), their time can be described as a linear function of the global time. This approach is used in the tracing library of the SCALASCA toolkit [2], which performs offset measurements between all local clocks and an arbitrarily-chosen master clock once at program initialization and once at program finalization. However, as the assumption of constant drift is only an approximation, violations of the clock condition may still occur.

The *controlled logical clock* (CLC) [5], an enhancement of Lamport's logical clock [3], is a method to retroactively correct timestamps violating the clock condition. The algorithm requires timestamps with limited errors (achievable through weak pre-synchronization) and a globally-unified tracefile. Since modifying individual timestamps might dialate local time intervals and even introduce new violations, the correction considers the context of the modified event by stretching the local time axis in the immediate vicinity of the affected event. The *extended controlled logical clock* (ECLC) [1] extends the controlled logical clock to apply to collective communication to provide a more complete correction of realistic message-passing traces. In addition to broader applicability, the ECLC algorithm operates on distributed trace files and therefore scales to thousands of application processes.

3.1 The Clock Condition

A *clock condition violation* occurs if the receive event of a message has an earlier timestamp than its matching send event. That is, the *happened-before* relation $e \rightarrow e'$ [5] between two events e and e' with their respective timestamps $C(e)$ and $C(e')$ does not hold. A clock condition violation between two events is defined as:

$$\exists\, e, e' : e \rightarrow e' \wedge C(e) \geq C(e'). \tag{3.1}$$

The CLC algorithm restores the clock condition using happened-before relationships between distributed events derived from point-to-point communication event semantics. More precisely, if the

condition is violated for a send-receive event pair, the receive event is moved forward in time. This adjustment is called *forward amortization*. To preserve the length of intervals between local events, events immediately preceding the corrected event are moved forward as well. This adjustment is called *backward amortization*. A detailed description of the CLC algorithm and a review of further synchronization approaches can be found in [5], [6], and [1].

3.2 Forward Amortization

Forward amortization is the process by which timestamps are moved forward to maintain the clock condition. Timestamps computed by the CLC are denoted by the symbol LC'. LC' is modeled with t as the wall clock time and $T(t)$ as the global time to which the process clocks $C_i(t)$ $(i = 0..n-1)$ are synchronized. Next, n is the number of processes, e_i^j is the j^{th} event on process i and so $E = \{e_i^j | i = 0..n-1, j = 0..j_{\max}(i)\}$ is the set of all events in the trace. The set of matching send and receive pairs is defined with

$$M = \{(e_k^l, e_m^n) | e_k^l \ = \ \text{send event}, e_m^n \ = \ \text{matching receive event}\}. \tag{3.2}$$

Note that the send event always marks the beginning of a send operation whereas a receive event marks the end of a receive operation. e_i^j is an internal event if it is neither a send nor a receive event. δ_i is the minimal difference between two events on process i, and $\mu_{k,i}$ is the minimum message delay of messages from process k to process i. Finally, γ_i^j is a control variable with $\gamma_i^j \in [0, 1]$. For each process, LC_i' is defined as

$$LC_i'(e_i^j) := \begin{cases} \max(LC_k'(e_k^l) + \mu_{k,i}, \\ LC_i'(e_i^{j-1}) + \delta_i, \\ LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ C_i(t(e_i^j))) \quad \text{if } \underset{e_k^l}{\exists} (e_k^l, e_i^j) \in M \end{cases} \tag{3.3}$$

$$\begin{cases} \max(LC_i'(e_i^{j-1}) + \delta_i, \\ LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))), \\ C_i(t(e_i^j))) \qquad \text{otherwise.} \end{cases} \tag{3.4}$$

As can be seen, the algorithm consists of two equations. Equation (3.3) adjusts the timestamps of receive events while Equation (3.4) modifies timestamps of internal and send events. Note that for each process, the terms $LC_i'(e_i^{j-1}) + \delta_i$ and $LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1})))$ must be omitted for the first event $(j = 0)$.

Through the term $C_i(t(e_i^j))$ in Equation (3.3) and Equation (3.4), the algorithm ensures that a correction is only applied if the trace violates the clock condition. The new timestamps satisfy the clock condition, since the term $LC_k'(e_k^l) + \mu_{k,i}$ in Equation (3.3) ensures that $LC'(e_i^j)$ is put forward compared to $C_i(t(e_i^j))$ if needed in case of a clock condition violation. To ensure that the clock does not stop after a clock condition violation, the term $LC_i'(e_i^{j-1}) + \gamma_i^j (C_i(t(e_i^j)) - C_i(t(e_i^{j-1}))$ in Equation (3.3) and Equation (3.4) approximates the duration of the original communication after a clock condition violation. Rabenseifner describes the control mechanism and γ_i^j in more detail in [6].

3.3 Backward Amortization

Backward amortization is applied to smooth jump discontinuities caused during forward amortization. This is done by distributing a jump of size Δt over an amortization interval L_A preceding the violating receive event by using a process-local, piecewise linear correction. In Figure 3.1, the horizontal axis represents LC_i^b, which is equal to LC_i' (i.e., the state after forward amortization) but without the jump Δt at event r. The vertical axis shows offsets to LC_i^b after applying different stages of backward amortization.

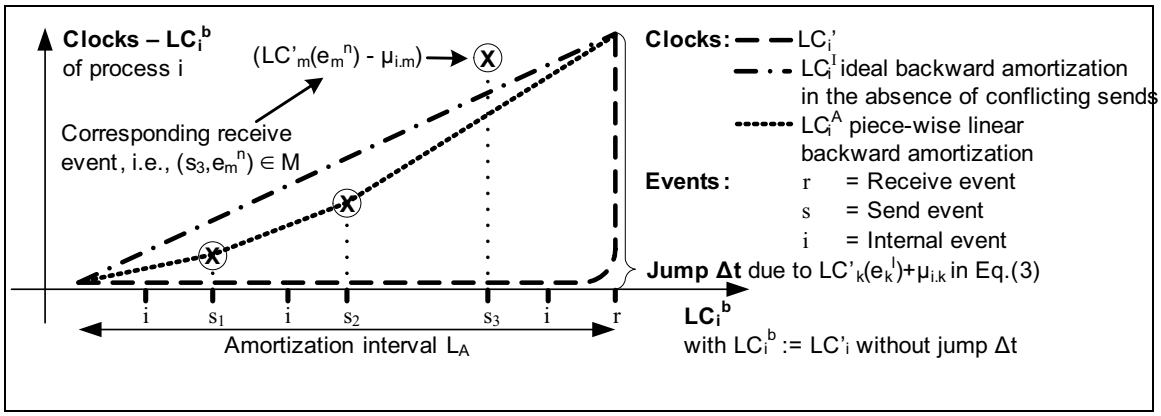

Figure 3.1: Algorithm for backward amortization.

In order to preserve the clock condition, the correction must not advance the timestamps of send events farther than $LC'_m - \mu_{i,m}$ of the corresponding receive event e_m^n of a process m. These upper limits are shown as circled values above the locations of the send events in Figure 3.1. If these limits are smaller than the dashed-dotted line (here at events s_1 and s_2), then a piecewise linear interpolation function must be used, represented as a dotted line in Figure 3.1. If there are no violating send events in the backward amortization interval of a process i, then an ideal linear interpolation can be used (the dash-dotted line in Figure 3.1). For each receive event with a jump, the backward amortization algorithm is applied independently. If there are additional receive events inside the amortization interval during such a calculation step, then these events can be treated like internal events, because advancing the timestamp of a receive event further cannot violate the clock condition.

3.4 Collective Operations

A single collective operation can be considered as a composition of many point-to-point communications. That is, if S and R denote the set of send and receive events in a collective operation instance i, respectively, then for each call to a collective operation, the set of all send-receive pairs M is enlarged by adding $S \times R$.

1-to-N: One root process sends its data to N other processes. Example are `MPI_Bcast`, `MPI_Scatter`, and `MPI_Scatterv`. S only contains the send event of the root process, whereas R contains receive events from all processes of the communicator with a data length greater than zero.

N-to-1: One root process receives its data from N processes. Examples are `MPI_Reduce`, `MPI_Gather`, and `MPI_Gatherv`. R only contains the receive event on the root process. S is the set of send events on all processes of the communicator with a data length greater zero. Given that the root process is not allowed to exit the operation until the last process enters the operation, the latest enter event is the relevant send event to fulfill the collective clock condition. Hence, if S contains more than one element, the term $LC'_k(e_k^l) + \mu_{k,i}$ in Equation (3.3) must be replaced by the maximum of $LC'_k(e_k^l) + \mu_{k,i}$ over all $e_k^l \in S$.

N-to-N': All processes of the communicator are sender and receiver. Examples are `MPI_Allreduce`, `MPI_Allgather`, `MPI_Alltoall`, and `MPI_Barrier` with N'=N, and the variable length operations `MPI_Reduce_scatter`, `MPI_Allgatherv`, and `MPI_Alltoallv`. S and R are all enter and collective exit events whose processes contribute input data or receive output data. For a call to `MPI_Barrier`, all processes of the communicator contribute to S and R.

Special cases: For `MPI_Scan` and `MPI_Exscan`, the set of messages added to M cannot be expressed as the Cartesian product $S \times R$. These cases are currently ignored by the ECLC algorithm and therefore are not handled in our implementation. This functionality is a proposed future work.

Chapter 4

Implementation

The ECLC algorithm was implemented as part of the SCALASCA performance analysis toolkit, so the implementation is described in terms of the SCALASCA event model. The Parallel Event Analysis and Recognition Library (PEARL) [2] provides the necessary classes and methods to process and operate on an event stream taken from a parallel trace file. For each individual event, SCALASCA records at least a timestamp, the location (i.e., the process) causing the event, and the event type. Depending on the event type, additional information may be supplied. The event model distinguishes between programming-model independent events, such as entering and exiting code regions, and events related to MPI operations. The latter include events representing point-to-point operations, and the completion of collective operations. *Collective exit* events are specializations of normal exit events carrying additional information (i.e., the communicator) that allows us to identify concurrent collective exits belonging to the same collective operation instance.

Because the PEARL encapsulation of SCALASCA events does not provide methods for determining the logical type (i.e. role) of an event in an event stream, a set of predicate functions to determine the logical type of an event, based on the event type and code region where the event was produced, were created. Every possible event is abstracted into one of three logical types: send, receive, and internal. For example, an "enter" event may be the start of a collective communication operation and therefore should be considered a "send" event when amortizing. This abstraction makes it possible to express collective operations as compositions of point-to-point operations. Table 4.1 gives the event sequences recorded for a few typical MPI operations.

Table 4.1: Event sequences recorded for a few typical MPI operations.

Function name	**Event sequence**
`MPI_Send()`	(enter, send, exit)
`MPI_Recv()`	(enter, receive, exit)
`MPI_Allreduce()`	(enter, collective exit) for each participating process

PEARL's parallel replay functionality is used to process each local tracefile in parallel, similar to the approach used by the SCALASCA tracefile analyzer (see [2] for details). The parallel replay approach has several important advantages. First, a single, globally-unified tracefile is not required, which greatly improves scalability. Secondly, it requires very little overhead, so machines with low per-process memory (e.g. IBM BlueGene/L) are supported. To perform a parallel replay, each process reads the local tracefile into memory and traverses the event stream with an instance of the PEARL `Event` iterator class. This approach requires exactly the same number of processes as was used in the original application and guarantees scalability equal to or greater than the original application scalability. For each communication event in the stream, the following algorithm is applied:

1. Determine the logical type of the event based on event type and code region
2. If the event is a logical "send" event:

   (a) Perform a local forwards amortization
   (b) Forward-replay the communication so the receiving process has the correct remote timestamp
   (c) Backwards-replay the communication and store the remote timestamp in a ring buffer
3. If the event is a logical "receive" event:
   (a) Forward-replay the communication so the sending process has the correct remote timestamp
   (b) Perform a local forwards amortization
   (c) Backwards-replay the communication and store the remote timestamp in a ring buffer
   (d) Detect any jump discontinuity and perform backwards amortization if required
4. If the event is a logical "internal" event: Perform a local forwards amortization

The control mechanism used for the extended controlled logical clock requires a global view of the trace data to calculate γ_i. Establishing a global view of the trace data is not feasible with the replay-based approach since communication would be required for every single event. This can be solved by performing multiple passes until the maximum error e is below a predefined threshold ϵ. In this implementation, γ is fixed at $\gamma = 0.99$, but implementing the iterative control mechanism is trivial and will be done in future work.

At this time, the SCALASCA cannot record measurements of communication latency and minimum time between local events. It is impossible to calculate per-process δ and μ without this information, so the current implementation used fixed values of $\delta = 1.0 \times 10^{-9}$ and $\mu = 1.0 \times 10^{-6}$. Extending SCALASCA and PEARL to make such measurements available is a topic of current research, and the ECLC implementation is designed to accept these values as parameters when they become available.

Listing 4.1: Using `Synchronizer` with PEARL

```
1  // Initialize global definitions
2  GlobalDefs* defs = new GlobalDefs(archive);
3  // Prepare the local trace
4  LocalTrace* trace = new LocalTrace(*defs, archive, rank);
5  // Make sure calltree IDs are unified
6  PEARL_mpi_unify_calltree(*defs);
7  // Preprocess event timestamps
8  trace->preprocess();
9
10 // Set up callbacks for events
11 Synchronizer sync(rank);
12 CallbackManager fwdManager;
13 fwdManager.register_callback(ANY, PEARL_create_callback(&sync, &
       Synchronizer::amortize));
14
15 // Perform the replay
16 PEARL_forward_replay(*trace, fwdManager, NULL);
```

Figure 4.1 shows the program class diagram. The `ControlledLogicalClock` class encapsulates the Controlled Logical Clock. The member variables `delta`, `mu`, and `gamma` correspond to δ, μ, and γ in Equations 3.3 and 3.4. The `value` member variable is the current clock value. The member functions `amortize_forward_intern` and `amortize_forward_recv` correspond to Equations 3.3 and 3.4, respectively. The definitions of these functions are shown in Listing 4.2. The `RingBuffer` class provides an optimized, light-weight, templated ring-buffer class to store remote timestamps for use during backwards amortization. Memory in `RingBuffer` is

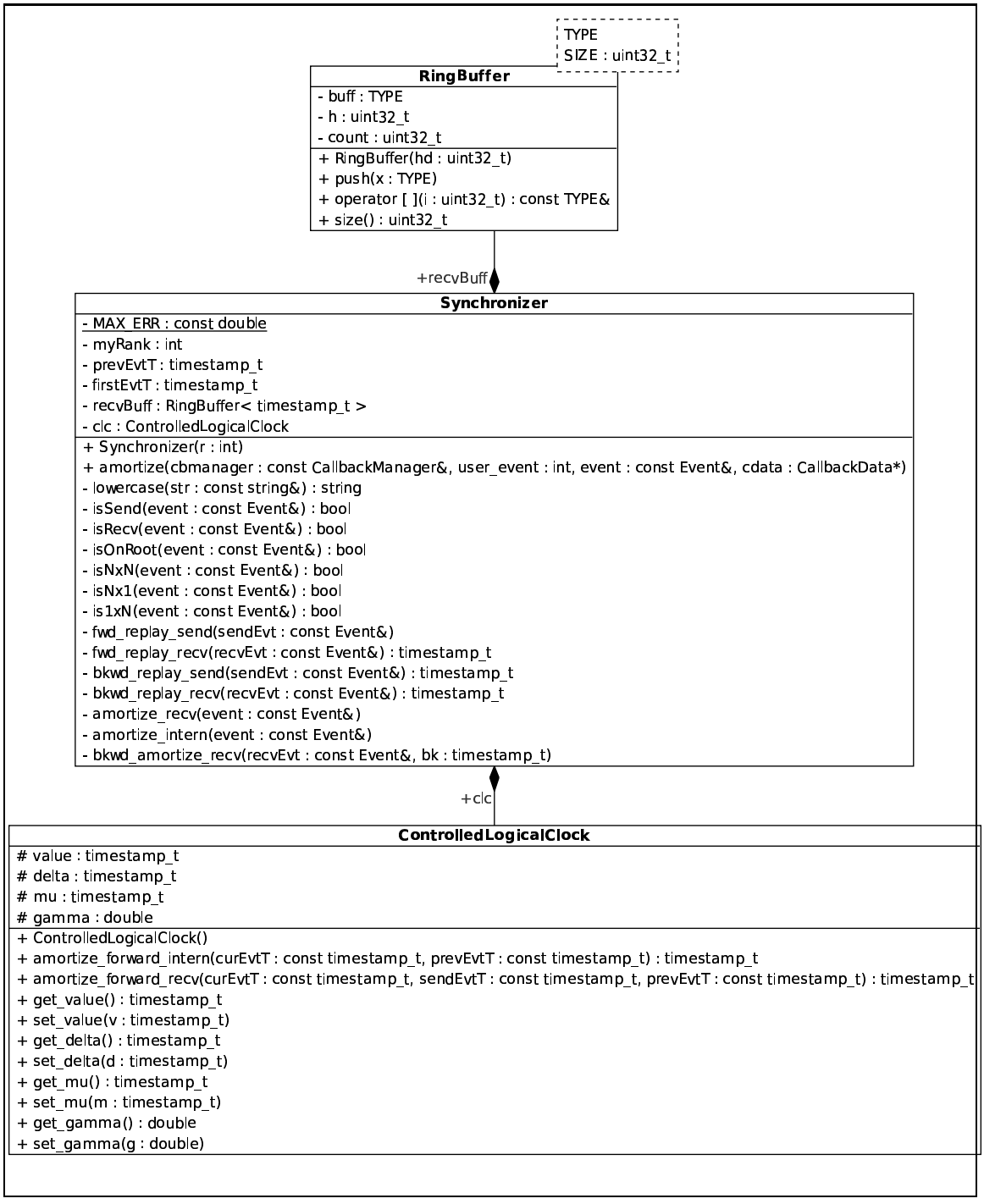

Figure 4.1: Program Class Diagram.

statically allocated, making this class both highly efficient and suitable for low-memory architectures. (Although dynamic memory allocation would normally be preferable in a restricted memory environment, we take advantage of the fact that the buffer will be filled to capacity for the majority of the program's execution time and so avoid the overhead of a dynamic data structure.) The `Synchronizer` class encapsulates the Extended Controlled Logical Clock. The `myRank` member variable records the current process rank (i.e. location), which is identical to the process rank of the application process which produced the local trace file. The `prevEventTime` and `firstEventTime` member variables corespond to $LC'_i(e_i^{j-1})$ and $LC'_i(e_i^0)$, respectively, in Equations 3.3 and 3.4. The `amortize` member function is a callback function for use with the PEARL replay mechanism, as shown in Listing 4.1. I discuss the other member functions in more detail in Chapters 4.1 and 4.2.

Listing 4.2: C++ implementation of Equations 3.4 and 3.3

```
1  timestamp_t amortize_forward_intern(timestamp_t curEvtT,
       timestamp_t prevEvtT) {
2    value = std::max(value + delta, std::max(value + gamma * (
       curEvtT - prevEvtT), curEvtT));
3    return value;
4  }
5
6  timestamp_t amortize_forward_recv(timestamp_t curEvtT,
       timestamp_t prevEvtT) {
7    timestamp_t internT = amortize_forward_intern(curEvtT, prevEvtT
       );
8    value = std::max(sendEvtT + mu, internT);
9    return internT;
10 }
```

When performing backwards amortization, both an LC_i^b which accounts only for local time axis dilation and an LC'_i which also considers remote timestamps are required. `amortize_forward_intern` and `amortize_forward_recv` set `value` to LC'_i and return LC_i^b, but in the case of Equation 3.4, $LC_i^b = LC'_i$.

4.1 Forward Replay and Forward Amortization

Table 4.2: Timestamps exchanged during forward amortization.

Type of operation	timestamp exchanged	MPI function
1-to-1	timestamp of send event	`MPI_Send`
1-to-N	timestamp of root enter event	`MPI_Bcast`
N-to-1	max(all enter event timestamps)	`MPI_Reduce`
N-to-N'	max(all enter event timestamps)	`MPI_Allreduce`

The first step in replaying a communication is the forward replay, which is performed by `fwd_replay_send` and `fwd_replay_recv`. Here the process which was the sender in the original application sends the timestamp of the send event to the process which was the receiver in the original application. Communication proceeds in the same direction as it did in the original application (i.e. forward). This is all the communication required to compute forward amortization, since only local timestamps and at most one remote timestamp is required. Depending on the type of the original communication operation, the timestamps are exchanged using different MPI function calls as listed in Table 4.2.

Forward amortization is performed by `amortize_recv` or `amortize_intern`, depending on if the event is a receive event or an internal event (recall that send events are considered internal events). `amortize_recv` invokes backwards amortization if a clock condition violation is detected after forward amortization has been performed.

4.2 Backwards Replay and Backwards Amortization

Table 4.3: Timestamps exchanged during backward amortization.

Type of operation	timestamp exchanged	MPI function
1-to-1	Implementationtimestamp of receive event	`MPI_Send`
1-to-N	min(all collective exit event timestamps)	`MPI_Reduce`
N-to-1	timestamp of root collective exit event	`MPI_Bcast`
N-to-N	min(all collective exit event timestamps)	`MPI_Allreduce`

The second part of a communication replay is the backward replay, which is performed by `bkwd_replay_send` and `bkwd_replay_recv`. Here the process which was the sender in the original application receives the timestamp of the receive event from the process which was the receiver in the original application. The roles of sender and receiver are reversed in reference to the original application, so the communication proceeds in a "backwards" direction. Depending on the type of the original communication operation, the timestamps are exchanged using different MPI function calls, as listed in Table 4.3.

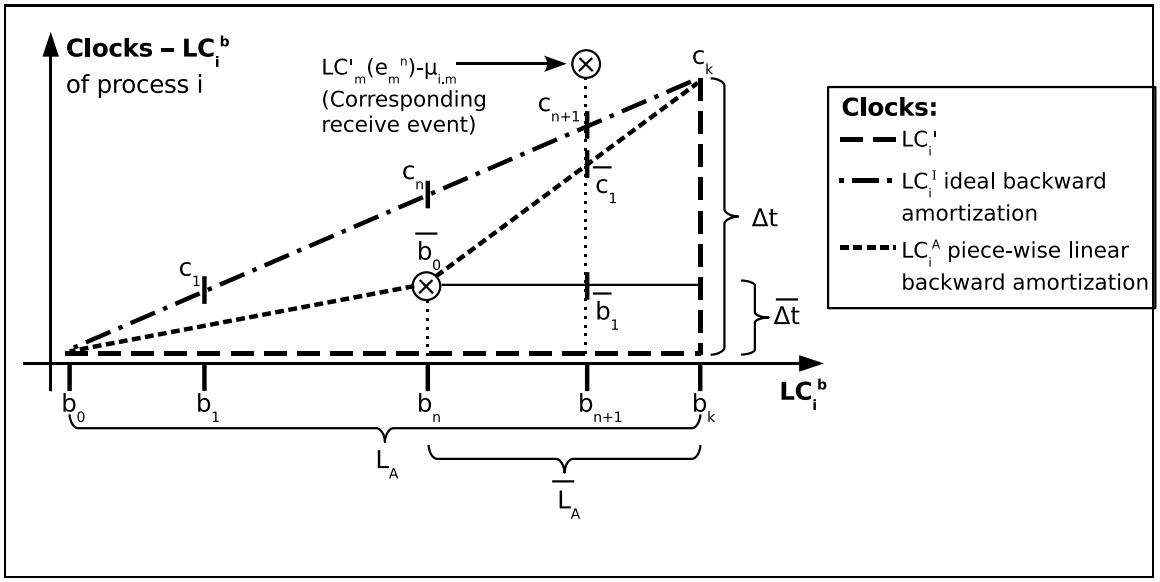

Figure 4.2: Datapoints Required During Backwards Amortization

Backwards amortization is performed by `bkwd_amortize_recv`. The datapoints required to calculate backwards amortization are described in Figure 4.2. The backward amortization algorithm is as follows:

1. Let e_n be an event preceding the event which caused the Δt timeline jump. ($timestamp(e_n) = b_n$)
2. Calculate the ideal linearly interpolated timestamp, c_n, for e_n.
3. Let $timestamp(e_n) := c_n$
4. If e_n is a send event and c_n is greater than the receive event timestamp for this send event, then a piecewise linear interpolation is calculated as follows:
   (a) Let c_n equal the receive event timestamp
   (b) Let $\bar{e_n}$ be the event directly following event e_n (i.e. e_{n+1})

(c) Calculate the piecewise linearly interpolated timestamp, $\bar{c_n}$, for $\bar{e_n}$

(d) Let $timestamp(\bar{e_n}) := \bar{c_n}$

(e) Let $\bar{e_n} := e_n \bar{+} 1$

(f) If $timestamp(\bar{e_n}) == c_k$, continue from Step 5. Otherwise, continue from Step 4a

5. Let $e_n := e_{n-1}$

6. If $timestamp(e_n) == b_0$, END backwards amortization. Otherwise, continue from Step 1.

Equations 4.2 - 4.8 describe how the datapoints in Figure 4.2 are calculated.

$$\Delta t = c_k - b_k \qquad \bar{\Delta} t = \bar{b_0} - b_n \qquad (4.1) \qquad (4.2)$$

$$L_A = b_k - b_0 \qquad \bar{L_A} = b_k - b_n \qquad (4.3) \qquad (4.4)$$

$$b_n = timestamp(e_n) \qquad \bar{b_n} = \frac{\frac{\Delta t}{L_A} b_0 + c_n}{\frac{\Delta t}{L_A + 1}} + \bar{\Delta} t \qquad (4.5) \qquad (4.6)$$

$$c_n = \frac{\Delta t}{L_A}(b_n - b_0) + b_n \qquad \bar{c_n} = \frac{\Delta t - \bar{\Delta} t}{\bar{L_A}}(\bar{b_n} - b_n) + \bar{b_n} \qquad (4.7) \qquad (4.8)$$

Chapter 5

Acknowledgements

Daniel Becker of Forschungszentrum Jülich is the primary architect of the Extended Controlled Logical Clock, which was first formulated in [1]. He oversaw my research in Jülich and was my main interface to the researchers and resources there. Prof. Dr. Felix Wolf leads the SCALASCA research and development group and provided me with every opportunity to improve my experience in Jülich. Ágnes Vajda, Chris Medalis, and Vijay Renganathan were my CESRI contacts and excellent hosts. This work would not have been possible without Herr Jamie Bishop of the Department of Foreign Languages at Virginia Polytechnic Institute and State University.

Chapter 6

Biography

John C. Linford is a PhD student of computer science at Virginia Tech. He received his Bachelor's of Computer Science and Mathematics in 2005 from Weber State University in Ogden, Utah. John graduated from Weber State as the Crystal Crest Scholar of the Year, the school's highest academic honor, and was an adjunct professor of computer science before beginning his graduate studies at Virginia Tech. Under Dr. Adrian Sandu, John is studying high performance computing systems and advanced multiphysics models, such as air quality and weather models. John is a member of the Virginia Tech Triathlon Team and competed in the 2007 USA national triathlon championship. His interests include cooking, playing piano, and art.

Bibliography

[1] D. Becker, R. Rabenseifner, and F. Wolf. Timestamp synchronization for event traces of large-scale message-passing applications. In *Proc. 14th European PVM/MPI Conference*, Paris, France, September 2007. Springer.

[2] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr. Scalable parallel trace-based performance analysis. In *Proc. 13th European PVM/MPI Conference*, Bonn, Germany, September 2006. Springer.

[3] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[4] D. L. Mills. Network Time Protocol (Version 3). The Internet Engineering Task Force - Network Working Group, March 1992. RFC 1305.

[5] R. Rabenseifner. The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters. In *Proc. 5th EUROMICRO Workshop on Parallel and Distributed (PDP'97)*, pages 477–484, London, UK, January 1997.

[6] R. Rabenseifner. *Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen*. PhD thesis, Universität Stuttgart, March 2000.