TOWARDS ADAPTIVE CACHING FOR PARALLEL AND DISTRIBUTED SIMULATION

by

ABHISHEK CHUGH

(Under the Direction of Maria Hybinette)

ABSTRACT

We investigate factors that impact the effectiveness of caching to speed up discrete event simulation. Walsh and Sirer have shown that a variant of function caching (staged simulation) can improve the performance of simulation in a networking application (Walsh and Sirer 2003). Consider, however, that the effectiveness of a caching scheme depends significantly on cache size, the cost of consulting the cache, the cache hit rate, and the cost of completing the computation in the case of a cache miss. We hypothesize that adaptive techniques can be used to optimize caching parameters (e.g. cache size), and demonstrate an adaptive scheme that decides whether to utilize caching at an LP depending on observed cache performance and event processing times. This paper focuses on a quantitative evaluation of these relationships using our own caching implementation with the P-Hold synthetic workload application (Fujimoto 1990) running on the GTW simulation kernel (Das et al. 1994). Experiments show that as the cache size is increased, performance improves to a point, then degrades, and also that the adaptive technique can substantially improve speedup.

INDEX WORDS: Parallel and Distributed Event Simulation (PDES), Adaptive Caching, Caching Middleware.

TOWARDS ADAPTIVE CACHING FOR PARALLEL AND DISTRIBUTED SIMULATION

by

ABHISHEK CHUGH

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2004

© 2004

Abhishek Chugh

All Rights Reserved

TOWARDS ADAPTIVE CACHING FOR PARALLEL AND DISTRIBUTED SIMULATION

by

ABHISHEK CHUGH

Major Advisor: Dr. Maria Hybinette

Committee: Dr. Eileen T. Kraemer
Dr. Thiab Taha

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2004

DEDICATION

To my parents

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Maria Hybinette for her guidance, foresight, and her assurance in difficult times. Dr. Hybinette has been very generous with her time and wisdom. I would like to thank my parents and sister for their continuous encouragement and love. I would also like to thank Dr. Eileen T. Kraemer and Dr. Thiab Taha for their valuable suggestions and being a part of my committee.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS....................v

LIST OF FIGURES....................viii

CHAPTER

1 INTRODUCTION....................1

2 BACKGROUND....................3

3 RELATED WORK....................5

4 APPROACH....................9

1. CACHING MIDDLEWARE....................10

2. ADAPTIVE CACHING....................12

5 CACHE IMPLEMENTATION....................13

1. MIDDLEWARE....................13

2. TIME_SENSITIVE ADAPTIVE CACHING....................14

3. APPLICATION PROGRAMMER"S INTERFACE....................14

4. INITIALIZATION PHASE....................14

5. EXECUTION PHASE....................15

6. WRAPUP PHASE....................16

6 PERFORMANCE....................17

1. BASIC CACHING EXPERIMENTS: HIT RATES....................18

2. BASIC CACHING EXPERIMENTS: SPEEDUP....................19

3. ADAPTIVE CACHING EXPERIMENTS....................22

7 ADVANTAGES AND LIMITATIONS 24

8 FUTURE WORK 26

9 CONCLUSION 27

REFERENCES 28

LIST OF FIGURES

Page

Figure 1: Scenario demonstrating Lazy Re-evaluation 8

Figure 2: Caching implementation 10

Figure 3: Caching Middleware 11

Figure 4: Performance of P-Hold: Hit Ratio 18

Figure 5: Performance of P-Hold: Speedup 20

Figure 6: Performance of P-Hold: Speedup 21

Figure 7: Performance of P-Hold: Adaptive time sensitive caching 23

CHAPTER 1

INTRODUCTION

Redundant computations are a significant source of inefficiency in discrete event simulations. Redundant computations have several causes. In a typical simulation, events are processed in timestamp order regardless of the fact that similar computations might have been performed earlier. During a long simulation it is likely that identical events will recur, especially in repetitive or recursive applications. Processing the same events again during the course of a simulation will lead to a substantial number of redundant computations. Even when identical events do not recur, it is likely that the computations that make them up will be redundant. A number of approaches have been devised to address this problem.

In earlier work, we proposed simulation cloning as a means of reducing the number of redundant event computations in repeated sequential simulations (Hybinette and Fujimoto 2001). Repeated simulation is a means of evaluating the impact of different conditions or policies on the outcome of a real system (e.g. air traffic control systems). Cloning, however, does not address the problem of repeated computations within a single simulation run.

Caching is a mechanism for saving the results of expensive calculations for reuse later. If the cost of checking the cache is sufficiently low, overhead is negligible, yet the savings when a cache hit is successful can be great. Walsh has proposed simulation staging, a form of function caching, as a way to improve the performance of discrete event simulation in applications with a substantial number of redundant calculations (Walsh and Sirer 2003). His approach provides significant speedup (up to 40x in a networking application), but requires extensive structural revision of code at the user application level.

In this work we introduce caching middleware that resides between the application and kernel. This approach enables an application to take advantage of caching with only minor revision. Furthermore, no change is required at the kernel level. We use GTW, a distributed discrete event kernel. Applications

are implemented as set of Logical Processes (LPs) that exchange timestamped messages or events. When an LP processes an event, its state may change, and it may generate one or more events in response. Our middleware observes how the state of a logical process changes and the events it generates in response to a message. This information is cached for later reuse.

A separate cache is implemented for each LP. Because LPs may be distributed on different processors (or machines) separate caches can provide a significant advantage with respect to the cost of accessing the cache. It is also possible that individual caches can be smaller because individual LPs will explore only a subset of the possible space of LP states. There are advantages and limitations to our approach, which we explore experimentally.

Regardless of how caching is implemented for simulation (e.g. at the LP level, as staged simulation, or as function caching), there are a number of factors that will affect its utility. Namely, cache size and hit rate, the cost of checking the cache, and the cost of completing a computation in the case of a cache miss. We evaluate the impact of these factors on the performance of our caching mechanism using the P-Hold application running on GTW (Das et al. 1994).

We also present and evaluate adaptive caching. Observe that if the cost of checking the cache exceeds the cost of just doing the computation, caching will degrade performance. During a warm-up period adaptive caching gathers statistics on these costs; after the warm-up period the system may choose to skip caching if it is too expensive. We show that this approach can provide speedup beyond the performance of simple caching.

The next chapter covers background. Related work in is covered in chapter 3. Our caching approach is described in chapter 4. The implementation and the programming interface is described in chapter 5. Chapter 6 discusses performance results. Advantages and limitations are outlined in chapter 7 and we discuss future work in chapter 8. The paper closes with a conclusion and discussion.

CHAPTER 2

BACKGROUND

Computer simulation is computation that models the behavior of physical system, executing the model on a digital computer, and analyzing the execution output. Simulation, has become prevalent tool in almost every area of scientific research, e.g. in the area of physics, chemistry, telecommunication networks, forecasting & planning, and transportation systems (air traffic simulation) to mention a few. Computer simulations are a means to test assumptions, to observe the outputs or performance of a process or a system, and to test alternatives before implementing in the real world.

Computer simulation can be broadly classified as either continuous or discrete. In the continuous simulation model, the state of the simulation model changes continuously with time. In the discrete simulation model, state changes at discrete times. Discrete simulation is further divided into discrete time stepped and discrete event stepped simulations. Discrete time stepped simulations change state after some fixed time intervals. Discrete event simulation model assumes the system being simulated changes state at discrete points (and irregular time intervals) in simulated time. In discrete event simulation, the state of the simulation changes on occurrence of an event. An event in a discrete event simulation model corresponds to some instantaneous action in the physical system. In order to process events in the proper order events are associated with time stamps. On processing an event, new events having larger time stamps may be generated.

Parallel and distributed discrete event simulations (PDES) enable sequential simulations to run faster and become more fault-tolerant. In PDES a logical process (LP) models some part of the actual physical system. These LPs are mapped to physical entities (residing on processors). LPs communicate by exchanging time stamped messages.

A primary concern of PDES is that of synchronization. A parallel and distributed discrete event execution should yield the same result as a sequential execution of the same simulation program. To

ensure consistency between a sequential and parallel execution of a simulation program, LPs must processes events time stamp order. PDES commonly uses a conservative or optimistic synchronization protocol to accommodate this.

In conservative synchronization protocols each LP strictly avoids processing events out of time stamp order. It enforces consistency by avoiding events in the past. Each LP maintains a first in first out (FIFO) queue for each LP it communicates with. If one of the queues becomes empty, the LP blocks. The LP is blocked until another message arrives at the empty queue. We cannot process the events residing in other queues because an event might arrive on the empty queue having lower time stamp. Conservative protocols are prone to deadlock (a cycle of empty queues). There are many algorithms to avoid the deadlock such as the null message approach (Chandy and Misra 1978, Bryant 1977).

The optimistic synchronization protocol allows an event to be processed in the past, but recovers via a rollback mechanism (Jefferson and Sowizral 1982). Here, an LP processes events optimistically assuming there are no causality errors and processes events as they are received. If an event is received with a time stamp lower than previously processed event(s), the LP rolls back by restoring its state in simulation time just prior to the time stamp of the arrived message. The message that triggered the rollback is called a straggler message. It is necessary to cancel messages that were processed subsequent to the straggler, as they now may be incorrect.

Our research concerns increasing the efficiency of parallel and distributed discrete event simulations. The approach that we are proposing is appropriate for both optimistic and conservative simulations.

CHAPTER 3

RELATED WORK

Different techniques for reusing computations have previously been proposed and implemented. Memorization or function caching is a technique where inputs and the corresponding results are cached. The results can be reused under appropriate circumstances for same set of input arguments. This technique has been around for over 40 years (Bellman 1957; Michie 1968). Functional caching is widely used to increase the efficiency of programs, applications include: dynamic programming, incremental computations and sequential and parallel discrete event simulations.

Functional Caching in Dynamic Programming

Dynamic programming, due to the nature of the algorithm, makes repeated request for same function values. These function values can be cached and used later for same set of inputs. The repeated re-computations are thus saved.

Functional Caching in Incremental Computations

Incremental computation is a technique that takes advantage of repeated computations on inputs that are similar. It makes use of previously computed results in computing a new output.

Function caching can be used to obtain efficient incremental evaluations (Pugh and Teitelbaum 1989). A problem is broken down into sub-problems so that similar problems which share common sub-problems can reuse the cached sub-problems. A stable decomposition, decomposition scheme that decomposes two similar problems in a way that they share common sub-problems, is required for the functional caching to deliver efficient results.

Deriving incremental programs and caching intermediate results provides framework for program improvement (Liu and Teitelbaum 1995). It takes advantage that the intermediate results computed in a previous iteration can be re-used for computing the results of next iteration. It uses a three stage method to identify, use and maintain the intermediate results. The first stage breaks a function into sub-functions and gets all the intermediate results. In the second stage an incremental version of sub-functions are obtained under an input change. In stage three, useful intermediate results are extracted.

Functional Caching in Sequential Discrete Event Simulation

In discrete event simulation, staged simulation (Walsh and Sirer 2003) extends function caching to increase the efficiency of sequential simulations. It splits a large computation into smaller sub-computations. These sub-computations are then cached. Using caching at functional or sub functional level however, makes the approach heavily application dependent as knowledge of the computation and is dependencies are required to break it into sub-computations.

Staging assumes that the time to do the actual computation exceeds the overhead of the cache. Staging has been demonstrated in a simulation that models wireless networks, here it exploits grid-based neighborhood computations. In grid based computation the physical space is divided in to grids or buckets. The physical processes, which are in same bucket, can share common sub-computations. It is not clear how the benefit of Staging generalizes to other applications.

Challenges in functional and sub-functional (and staging) caching include how to split computations effectively due to interdependencies of input parameters or complexity of the function computation.

Functional Caching in Parallel Discrete Event Simulation

In cloning (Hybinette and Fujimoto 2001) simulations cloned at decision points share the same execution path before the decision point and thus only perform those computations once, after the decision point simulations can further share computations as long as the corresponding computations across the different simulations are not yet influenced by the decision point.

Updateable simulation proposed by (Ferenci et al. 2002) updates the results of a prior simulation run, called the base-line simulation, rather than re-executing a simulation from scratch. A drawback of this latter approach is that one must manage the entire state-space of the baseline simulation. Both of these mechanisms are appropriate for multiple similar simulation runs.

Another related approach, lazy re-evaluation a technique to reduce cost of rollback for optimistic simulation, caches the original event in anticipation that it will be re-used after the rollback and consequently avoid re-computation (West 1988). Lazy re-evaluation avoids re-computation after a rollback by comparing the current state with cached state prior to the rollback. An example and its motivation of this approach are given in Figure 1.

Figure 1(a) shows an event arrived at 4:00. It leads to an event at 5:00 and a message 'X' is sent to another LP to be processed at 7:00. Suppose a straggler arrives with time stamp 1:00. (Figure 1(b)). This leads to rollback of event processed at 4:00. But consider what if the event at 1:00 does not have any effect on the state at the time when event at 4:00 is processed i.e. the state during re-execution of event (after rollback) at 4:00 is same as what was when originally the event was processed. We are repeating the same computation.

In lazy-reevaluations the state information before the rollback is not discarded. While re-executing the rolled backed events, the state is compared with the state cached prior to rollback. If they are equal the reprocessing of the event can be avoided. In the example above we avoided the reprocessing of events at 4:00 and 5:00 in simulation time. The re-processing of a series of event computations which have been rolled back due to the arrival of straggler message can be saved. It is also known as jump forward optimization as we can jump forward a series of rolled back events without processing them if they are identical to when originally processed.

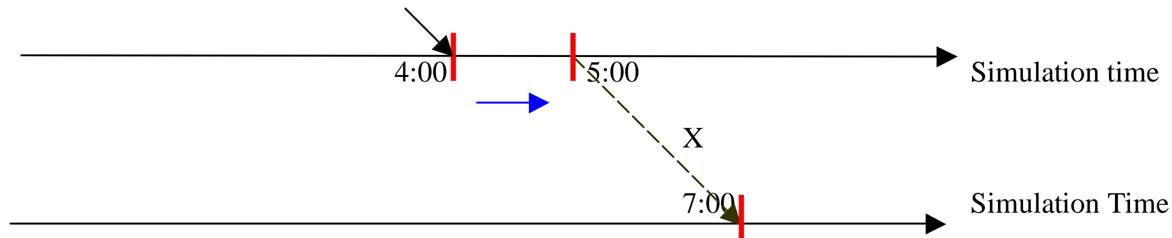

Figure 1(a): Scenario demonstrating Lazy Re-evaluation.

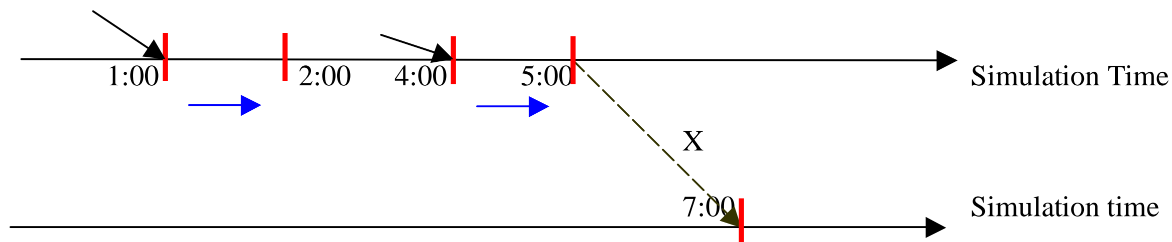

Figure 1(b): Scenario demonstrating Lazy Re-evaluation.

As in lazy re-evaluation we cache the event computations and avoid re-computing same events again to increase the efficiency. In lazy re-evaluation the process of caching and reusing events goes at kernel level to optimize optimistic simulation engine. Our cache works as a middleware between the simulation kernel and application. Event computations are stored in the cache during the execution phase. Before processing the event the cache is looked up and event computation is avoided in case of a hit. Our approach however is not limited on re-using event in the face of rollback. It is independent of the synchronization mechanism and works for both conservative and optimistic simulation engines.

CHAPTER 4

APPROACH

In order to evaluate the affect of various caching parameters on performance, we implement our own caching scheme and evaluate it experimentally. Our technique uses a distributed cache to store the results of event computations at each LP, where each LP maintains its own cache independently. The cache is indexed by the current state of the LP and the incoming event. The resultant state and output message are stored as results in the cache. The cache is implemented as a hash table that uses separate chaining to resolve collisions, e.g. the table is implemented as an array of linked lists (See Figure 2). The index or keys of the hash is computed from the contents of the current state of the LP and the arrival message. Resultant state and output message(s) are stored as results. The memory required for nodes of the link lists is allocated from a pre-allocated memory pool. Once the size of cache on a LP grows beyond the maximum allowed size (an adjustable and tunable parameter), previously cached results are replaced. The current cache replacement strategy simply replaces the entries that were stored the earliest in the cache for that particular index.

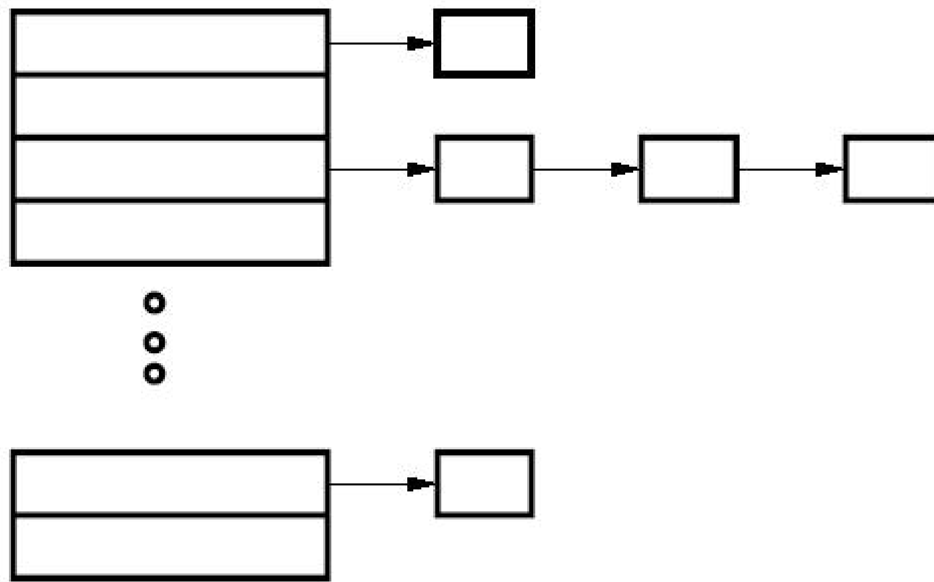

Figure 2: Caching Implementation: The cache is implemented as an array that is indexed as a hash function. The hash table uses separate chaining to resolve collisions and is implemented as an array of linked lists.

1. CACHING MIDDLEWARE

In our implementation the caching software is middleware independent of the simulation engine and the application. The approach can be used with both conservative and optimistic simulation engines. No changes to the underlying kernel are required, but a few calls must be added in the application code. However, we emphasize that no significant structural changes at the application level are necessary.

We provide an API for the user application to the middleware. Figure 3 shows how communication takes place between application and the kernel through the middleware. When the kernel attempts to deliver an event to the application code, the caching software intercepts it.

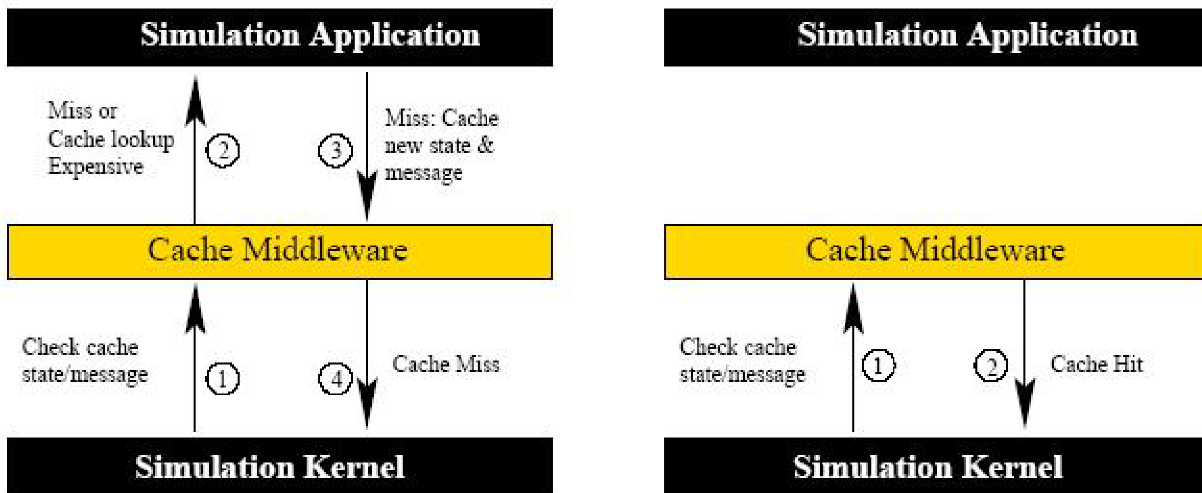

Figure 3: Caching Middleware: Our caching scheme is implemented as middleware between the simulation application and the simulation kernel. The left diagram illustrates the sequence of events in the case of a cache miss. The diagram on the right shows a cache hit.

The cache of the LP for which the message is intended is consulted. In case of a hit, the retrieved resultant state and message is passed back to the kernel (without the need to consult the application code). The LP's state is updated and the resultant event is scheduled by the kernel. In case of a miss the message is passed on to the application and event computation is performed. The resultant message and state information is captured by the middleware, where an entry is made into the cache for future reference. The message is then sent to the required LP through the kernel. If the cost of consulting the cache is small we can save significant computation in the case of a cache hit. In case of a miss the normal procedure of computation is performed and the results are cached. (Performance is evaluated in a later section). If the size of cache grows beyond the maximum allocated size per LP, results are overwritten on the previous cached entry. The cache overwrites the least recently used entries first. The middleware is not part of the kernel, so rollbacks do not have to be addressed at that level. Thus the approach can work with conservative and optimistic simulation engines.

2. ADAPTIVE CACHING

An advantage of a middleware implementation is that the middleware can evaluate the time required for an LP to perform an event computation. The middleware can also evaluate the overhead of caching, and make a determination as to whether it is better to just allow the LP to perform the computation or to use caching. The caching middleware does not reference the cache for very small event computations, but as the granularity of a computation becomes large the cache is referenced to improve performance, note however that maintaining the cache is more expensive in the beginning of the simulation since the cache is not warm. In our current implementation we switch to caching when processing time become more than some multiple of the caching overhead time. This is a tunable parameter, and may be set as a factor of the size of the event computation and size of the state. We cover details of the implementation next.

CHAPTER 5

CACHING IMPLEMENTATION

1. MIDDLEWARE

The caching middleware is independent of the simulation engine, and therefore does not require any changes to the underlying kernel. A few calls must be added at the application level, however. The user application and simulation kernel interact through function calls that are implemented in the middleware. The middleware thus intercepts calls between the two levels. A separate cache is maintained for each LP. In distributed or parallel simulations, the cache is correspondingly distributed or parallel.

As previously mentioned, the cache is implemented using a hash table. A hash index is computed using the contents of the present state of the LP and the current message to process. In the case of collisions (two different 3 state/message pairs map to the same index), records are appended to a linked list at the corresponding index location. When items are added to the cache, additional memory is allocated as needed. Then the size of the cache reaches a predetermined limit, a earliest-stored-first policy is used to free memory for reuse.

Unless noted otherwise, in the experiments described below the size of the hash table is set to 600 and the allocated memory will accommodate 14 all possible state/event pairs.

Memory operations such as allocate() and free() are expensive, especially if they are implemented as system calls. For efficiency we would like to avoid these calls if possible during simulation run. Accordingly, a custom memory management system is implemented whereby all memory allocation for caching is completed at the start of a run. A memory pool is created at initialization. The cache for each LP is also initialized at this time, but no memory is allocated to it.

2. TIME-SENSITIVE ADAPTIVE CACHING

The overhead associated with caching includes hashing, retrieving results and adding new event computation results. Our system tracks the time spent on caching and the time spent on actual computation. If caching becomes more expensive than the actual computation we stop using the cache. In this case the simulation application runs as if there is no caching middleware involved.

Our adaptive caching mechanism is implemented, by monitoring the cache overhead and the time to execute the event. Currently we store as execution time the last time the event was processed, and the overhead of the last time we accessed the cache (without processing the event). Before accessing the cache we compare the difference between the computation time and cache overhead. If it takes longer to process the event than to reference the cache (times some multiple) we reference the cache. 4.3

3. APPLICATION PROGRAMMER'S INTERFACE

Three functions are available to the simulation application. We list the functions' names below and describe them in detail later. The API functions are:

```
int cacheInitialize( int argc, char ** argv )
cacheStruct* cacheCheckStart()
cacheStruct* cacheCheckEnd()
void cacheCleanup()
```

These API functions are used during different phases of simulation run. We view state of the simulation as moving through three phases: initialization, execution, and wrapup. These phases are described in more detail below.

4. INITIALIZATION PHASE

To initialize caching, void **cacheInitialize()** is called during the the initialization phase of the simulation. The function has two arguments: argc and argv that specify command line parameters for caching. This sets up data structures that provide a memory pool for hashing states and inputs and initializes the caching hash tables. The command line arguments set limits on memory to be allocated and the initial size of the

cache. No system memory allocation is performed after initialization phase; our middleware administers its own memory pool. An example initialization is shown below:

```
void InitilizationPhase( int argc, char ** argv )
{
      /* other application initilization code is defined here */
      cacheInitialize( argc, argv );
}
```

5. EXECUTION PHASE

During the execution phase, **cacheCheckStart()** and **cacheCheckEnd()** are used to "wrap" the code used by the LP to respond to an event. These calls enable the middleware to measure the time required to execute the computation, or to return a cached result if appropriate. **cacheCheckStart()** returns NULL if the LP should execute its own computation, otherwise it returns new state information and the LP can skip its computation. In the adaptive approach, the middleware may return NULL even if the result is available in the cache, because it may have concluded that the computation is so inexpensive that it is cheaper than consulting the cache.

In case of a miss the event computation is performed. **cacheCheckEnd()** passes the new state, and any messages that were sent to the middleware to be saved in the cache. An example use of these calls is below:

```
void Event_Handler( event ) /* LP event processing */
{
      retval = cacheCheckStart( currentstate, event );
      /* cache miss, or caching expensive */
      if( retval == NULL )
      {
            /* original LP code */
            /* compute new state and events to be scheduled */
            /* allow cache to save results */
```

```
        cacheCheckEnd( newstate, newevents );
    }
    else
    {
        newstate = retval.state;
        newevents = retval.events;
    }
    schedule( newevents );
}
```

These calls enable the cache middleware to monitor time spent on actual event computation and caching overhead. The monitoring is transparent to the user application. The timer starts at the entry of **cacheCheckStart()** and ends with the call to **cacheCheckEnd().**

6. WRAPUP PHASE

When the simulation is complete **cacheCleanup()** is called to free the data structures and memory pool. It is called after simulation code is completed and before terminating the program. It returns on success and 0 otherwise. It does not take any input argument.

CHAPTER 6

PERFORMANCE

Caching efficiency depends on at least three features of the application being simulated: cost of event computations, running time of the simulation, and size of the state. Other factors include parameters of the caching scheme, which, in turn, affect how quickly the cache can be consulted. In general we expect better performance from caching as the cost of event computation increases, and worse performance as caching becomes more expensive.

There are a few other issues to consider as well. At initialization time, the cache is empty – and therefore not at all effective. However, as the cache "warms" up performance improves. Accordingly, longer simulations are more likely to benefit from caching. The size of the state is also important because for a given cache size, the number of event result computations stored is inversely proportional to the size of the state.

Quantitative results were obtained using two applications: Ping and P-Hold. Both the applications were run on GTW, an optimistic time warp simulation kernel (Fujimoto 1990; Das et al. 1994). Ping is a simple application where a single message is passed between LPs.

P-Hold provides a synthetic workload using a fixed message population. Each LP is instantiated by an event. Upon instantiation, the LP schedules a new event. The destination LP is chosen randomly. Note that while caching the resultant state, we remove the time stamp and store rest of the information. Similarly while comparing the state information during cache look up, we do not take timestamp into consideration. Evaluation of caching performance was conducted on an SGI origin 2000 with sixteen 195 MHz MIPS R10,000 processors. The unified secondary cache is 4 MB. The main memory size is 4 GB.

Three types of experiments were performed: 1) Experiments as proof of concept of the basic caching technique (no adaptive caching), 2) Experiments to evaluate the impact of cache size and

simulation running time on speedup for basic caching, and 3) Experiments to study the benefit of adaptive caching with regard to the cost of event computation. As proof of concept we calculated the increase in hit percentage versus the size of the cache, and simulation running time.

1. BASIC CACHING EXPERIMENTS: HIT RATES

As a proof of concept, we evaluated cache hit ratio versus the running time of the simulation and cache size using Ping application with 16 LPs. The plots in Figure 4 show that hit ratio generally increases as we increase the length of the simulation.

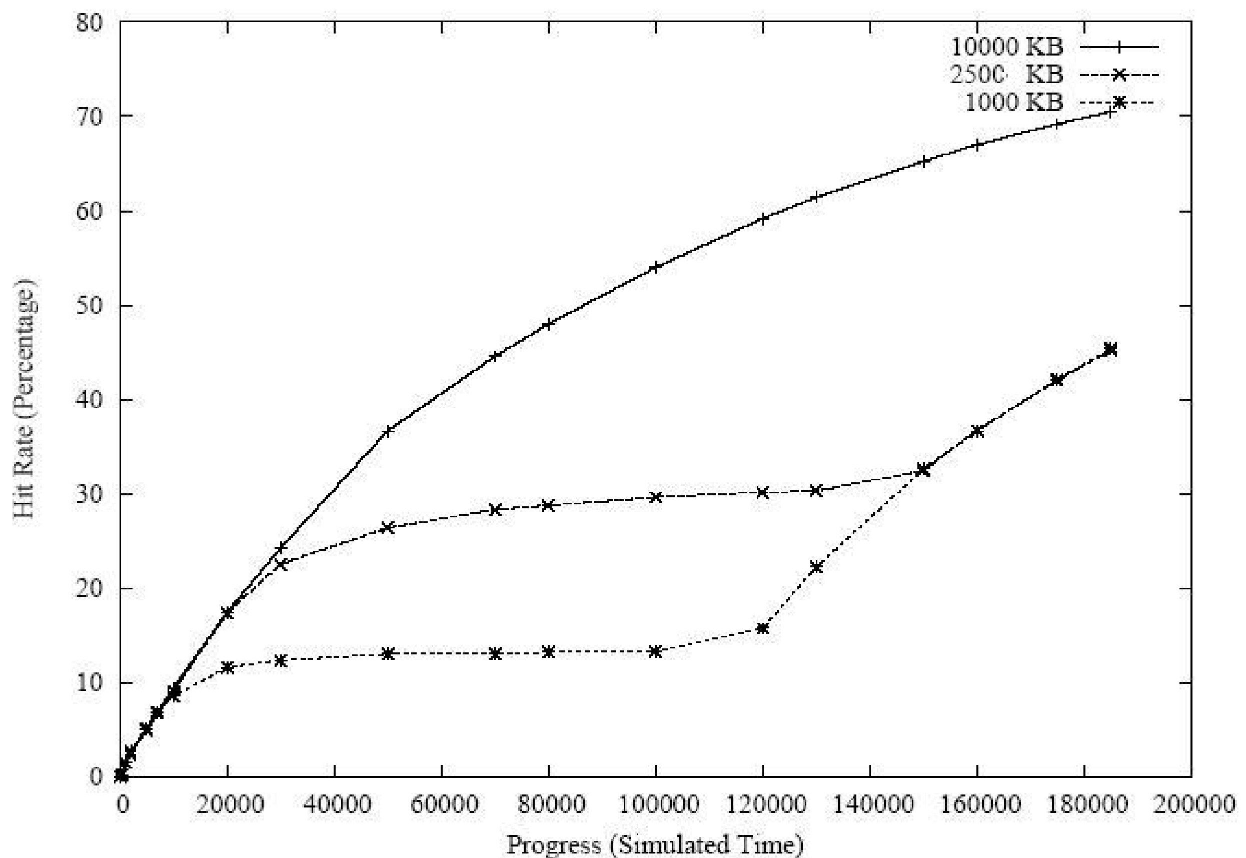

Figure 4: Performance of Ping: The hit ratio increases as the simulation progresses. Larger numbers indicate better performance.

Three experiments were run, using different cache sizes: a) cache size same as the size required to store all results (10 MB), b) cache size one-fourth the size required to cache all results (2.5 MB), and c) cache size one-tenth of the size required to cache all results (1 MB). As one would expect hit ratio also

increases as the cache size increases. Note that for case b) and c), hit rate performance levels off after 50,000 time units, then begins to improve after about 125,000 time units. We are not certain why this happens, but we speculate that this is a reflection of the "warm up" time for smaller caches.

Note that the hit rate sets an upper bound for speedup using caching. For instance, a hit rate of 50% would force us to complete 12 of the event calculations, limiting speedup to no more than 2.0 (assuming that the cost of checking the cache is negligible in comparison with the cost of completing the actual event computation). For the Ping application our hit ratio approaches 70%, thus forcing 30% of the computations, leading to a speedup limit of about 3.3.

2. BASIC CACHING EXPERIMENTS: SPEEDUP

We investigate a number of experiments to study the impact of caching on Ping and P-Hold performance. For comparison, a parallel simulation without caching is evaluated against the simulation with caching. Performance is evaluated as speedup of the run time of simulation using caching versus non-caching. Speedup is the running time of simulation not using caching divided by the running time of simulation using caching. Running time does not include the execution time of initialization and wrap-up phase. The experiments were run on 32 LPs mapped on 4 processors. The message population used is 640.

Figure 5 shows speedup versus the size of the cache for simulation having event computation time of 5 milliseconds. The simulation was run for 20,000 time units which included 580,000 - 640,000 processed events.

Speedup improves as the size of the cache is increased. However, beyond a certain point as cache size is increased, speedup declines. The maximum speedup in terms of cache size is attained approximately when the size of cache fit all result computations (7,299,072 Bytes). If the size of cache is increased further it becomes more of an overhead and the speedup decreases. In real world applications, the size of cache will be less than that required to store all possible results.

The maximum speedup expected was 2.32 (56.8 % event computation results reused). As we can see in the graph we get a maximum speedup of 3.2. We account the reason for increase in speedup beyond the theoretical limit to re-usage of event computations during reprocessing of events after rollback mechanism.

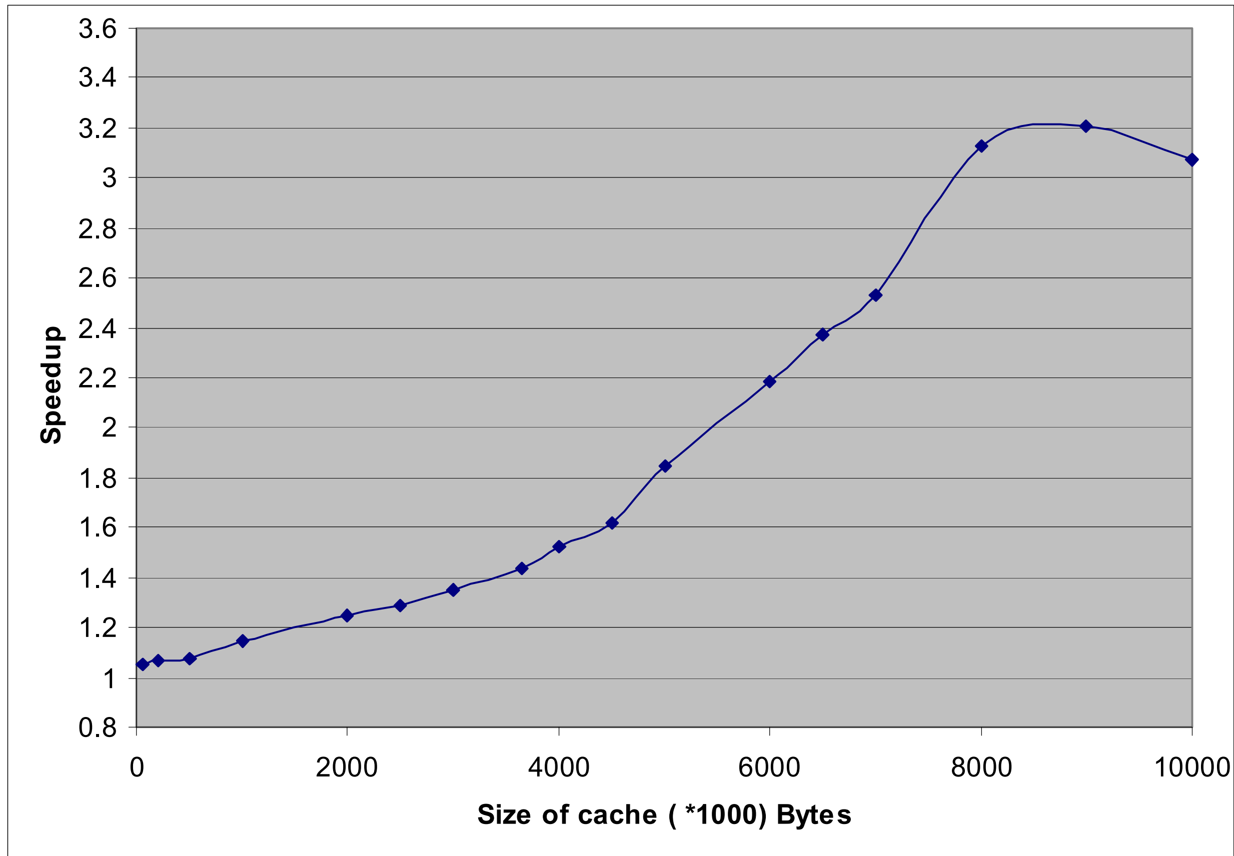

Figure 5: Performance of P-Hold: Speedup in comparison to traditional simulation (without caching) with respect to: the size of the cache. Event computation size is 5 milliseconds. Larger numbers indicate better performance.

Figure 6 shows speedup versus the size of the cache for simulation having event computation time of 3 milliseconds. The simulation was run for 25,000 time units which included 725,000-748,000 events.

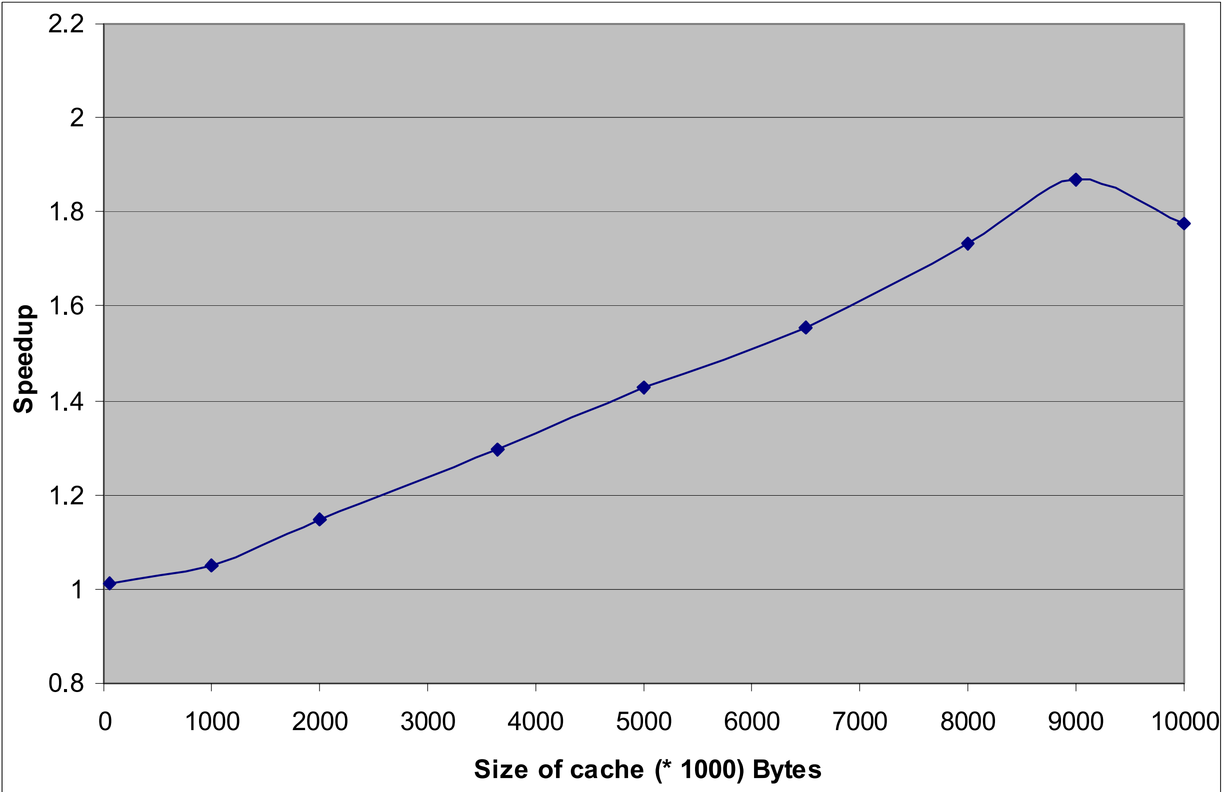

Figure 6: Performance of P-Hold: Speedup in comparison to traditional simulation (without caching) with respect to: the size of the cache. Event computation size is 3 milliseconds. Larger numbers indicate better performance.

The maximum theoretical expected speedup was 1.81 (44.87 % event computation results reused). We get a maximum speedup of 1.83. The speedup attained in this case is less (than expected from the previous speedup results for 5 milliseconds event computation time) as the interference of cache overhead increases as the size of event computation is reduced.

3. ADAPTIVE CACHING EXPERIMENTS

Observe that the performance of caching depends on many factors. In fact the impact of these factors on performance may change dynamically while the simulation runs. We suspect that adaptive techniques could be used to optimize caching parameters at run time. As an initial demonstration of this idea, we implemented a simple adaptive scheme, time sensitive caching (described above), and evaluated its performance.

The general idea is to track the cost of consulting the cache (which may change with time) in comparison to the cost of running the actual computation. If the computation cost exceeds a user defined multiple of the caching cost, the system chooses to use caching, otherwise it allows the application to compute the events, even if they are redundant.

We evaluated the approach by varying the cost of event computation from 0 to 3 milliseconds, then measuring performance for: a) a simulation without caching, b) a simulation using simple caching, and c) a simulation using time sensitive caching. Speedup was computed for simulation b) and c) in comparison to simulation a).

We conducted an initial experiment using the same application and caching parameters as in the experiments above. In this case we discovered that there was hardly any benefit to the adaptive technique until event computation costs were reduced to below 10 microseconds. The results were noisy and our ability to instrument the experiment over such small time intervals is limited. We suspect that the cost of caching in these conditions may be artificially low due to the large hash table and limited size of the state space.

In order to evaluate adaptive caching more fairly, we adjusted one of the parameters of the caching algorithm to make caching more expensive. In particular, we reduced the size of the hash table from 600 elements to 50 elements. This change forces a linear search for matches much more often. While a hash table of 50 elements may be artificially small, we believe that this change may more

accurately reflect the relationship between caching and computation costs in other simulation applications.

Figure 7 shows speedup results for simulations using simple caching and adaptive time sensitive caching. The adaptive algorithm was set to select caching when the cost of event computations exceeded a factor of ten of the caching cost. Notice that speedup for the adaptive technique is approximately 1.0 for event granularities of 0 to 2.0 milliseconds. In comparison, simple caching suffers a speedup ratio of 0.8 for very small event computation costs, and only improves to 1.0 when event granularity approaches 1.5 milliseconds. This means that the adaptive technique improves performance over simple caching in this region.

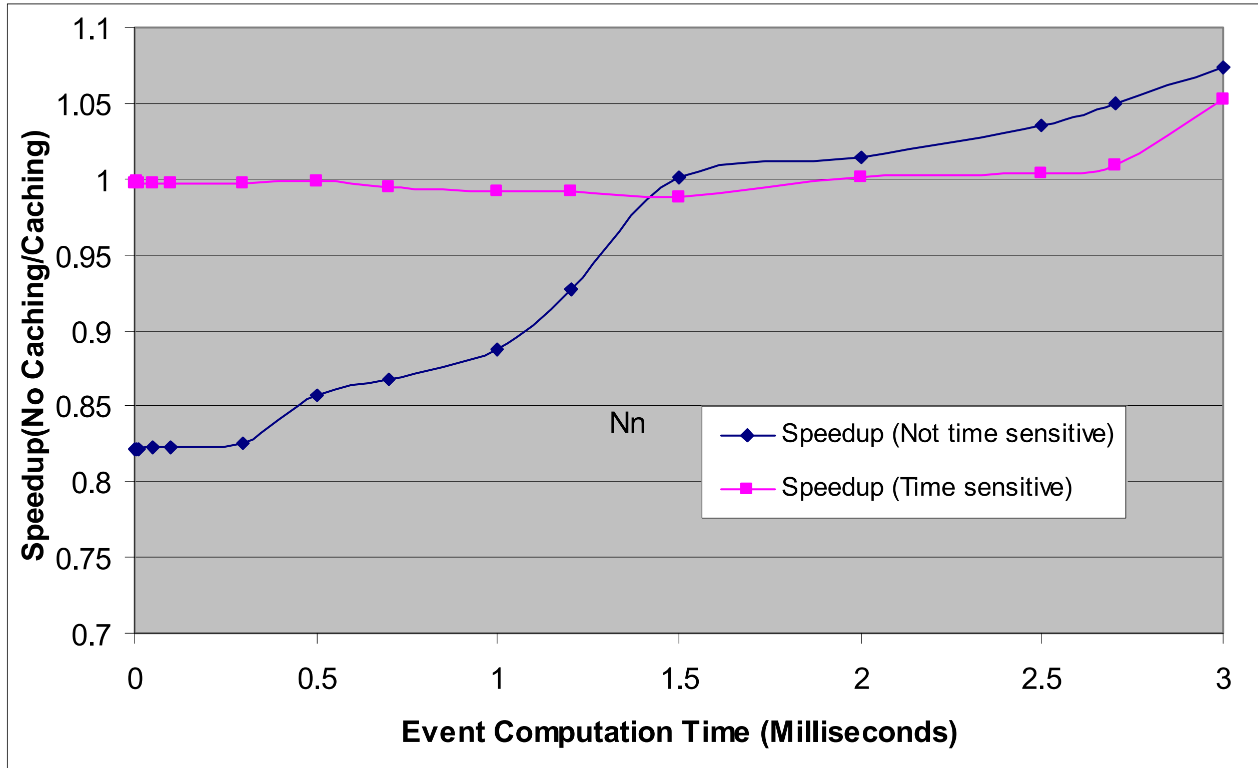

Figure 7: Performance of P-Hold: Shows speedup results for simulations using simple caching and adaptive time sensitive caching. Larger numbers indicate better performance.

As event granularity increases beyond 3 milliseconds, both simple caching and adaptive caching begin to exceed speedups of 1.05. In future work we will continue to evaluate the factors affecting performance in this region.

CHAPTER 7

ADVANTAGES AND LIMITATIONS

Our particular caching implementation offers several advantages, but suffers from some limitations as well. In any case, the focus of this work is to evaluate the effects of various caching parameters on simulation performance, and to look for opportunities to use adaptive techniques. These results should apply to any caching approach (e.g. middleware, function caching, or staged simulation).

The key advantages to our approach stem from the middleware implementation. In particular the simulation kernel requires no change, and the user application code requires only the addition of simple checkpoints. No major structural revision of the application code is necessary. From a user's point of view, integrating our caching scheme requires very little effort.

Performance results show that in the worst case our caching technique offers no speedup, but in the best case (for the P-Hold application) speedup approaches three. In comparison Walsh has reported speedups exceeding 40x (Walsh and Sirer 2003). Our speedup is limited primarily by the cache hit rate for the P-Hold application. Speedup, using any algorithm, will be limited to 3.33x when the hit rate is 70%. 40x speedups imply a hit rate of nearly 98%. The nature of our caching approach (that we cache on state/event pairs) will probably limit our hit rate, and thus speedup, in most applications.

The caching mechanism works effectively only when there are no side effects. If there is random information, (such as timestamp information or the result of a random number generator), in the results to be cached, the caching technique becomes ineffective. This is because the probability of reusing the computation becomes negligible.

The caching technique will be more effective for applications having smaller state size. If the size of the state is huge it will adversely affect the efficiency. Larger state size means more space will be occupied for each entry in to the cache resulting in fewer entries, thus reducing the probability of a hit.

Our results are based on experiments with the P-Hold application. Techniques involving caching do not inherently work efficiently with the randomness element involved. With randomness the number of event computation results possible will become unmanageable.

CHAPTER 8

FUTURE WORK

The future work involves adding additional adaptability, as in (Acar et al. 2002) to the caching mechanism i.e. integrating adaptive computing with coarse level functional caching. Adaptive functional programming maintains relationship between input and output as input changes. It keeps track of the input parameters, and therefore instead of re-evaluating the whole function from scratch, adaptive functional programming updates the output by reevaluating part of the program effected by changes in the input. The output is made adaptive to input by recording dependencies during initialization phase. Adding adaptability to the caching will improve efficiency but can make the technique proposed application dependent. We also like to fine tune our replacement strategy, and are considering similar approaches that are used in for functional caching as discussed in (Pugh 1988).

CHAPTER 9

CONCLUSION

We have investigated factors that impact the effectiveness of caching to speedup discrete event simulation. The key idea is enables an application to take advantage of caching with only minor revision. The overhead associated with caching includes hashing, retrieving results and adding new event computation results. Our system tracks the time spent on caching and the time spent on actual computation. If caching becomes more expensive than the actual computation we stop using the cache. In this case the simulation application runs as if there is no caching middleware involved.

Performance results show that in the worst case our caching technique offers no speedup, but in the best case (for the P-Hold application) speedup approaches three.

REFERENCES

ACAR, U. A., BLELLOCH, G. E., AND HARPER, R. 2002. Adaptive functional programming. ACM SIGPLAN Notices 37, 1 (Jan.), 247–259.

BELLMAN, R. E. 1957. Dynamic Programming. Princeton University Press.

DAS, S., FUJIMOTO, R., PANESAR, K., ALLISON, D., AND HYBINETTE, M. 1994. GTW: A Time Warp system for shared memory multiprocessors. In Proceedings of the 1994 Winter Simulation Conference Proceedings (December 1994), 1332–1339.

FERENCI, S. L., FUJIMOTO, R. M., AMMAR, M. H., AND PERUMALLA, K. 2002. Updateable simulation of communication networks. In Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS-2002) (May 2002), 107–114.

FUJIMOTO, R. M. 1990. Performance of Time Warp under synthetic workloads. In Proceedings of the SCS Multiconference on Distributed Simulation, Volume 22 (January 1990), 23–28. SCS Simulation Series.

HYBINETTE, M. AND FUJIMOTO, R. M. 2001. Cloning parallel simulations. ACM Transactions on Modeling and Computer Simulation (TOMACS) 11, 4, 378–407.

LIU, Y. A. AND TEITELBAUM, T. 1995. Caching Intermediate Results for Program Improvement. In ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (La Jolla, CA, June 1995), 190–201. ACM Press.

MICHIE, D. 1968. “memo” functions and machine learning. Nature, 19–22.

PUGH, W. 1988. An improved replacement strategy for function caching. In Proceedings of the 1988 ACM Conference on Lisp and Functional Programming (July 1988), 269–276. ACM: ACM.

PUGH, W. AND TEITELBAUM, T. 1989. Incremental computation via function caching. In Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (Austin, Texas, Jan. 11–13, 1989), 315–328. ACM SIGACTSIGPLAN: ACM Press.

WALSH, K. AND SIRER, E. G. 2003. Staged simulation for improving scale and performance of Wireless network simulations. In Proceedings of the Winter Simulation Conference, New Orleans, LA, December 2003.

WEST, D. 1988. Optimizing Time Warp: Lazy rollback and lazy re-evaluation. M.S. Thesis, University of Calgary.

CHANDY, K. AND MISRA, J. Distributed Simulation: A Case study in the design and verification of Distributed programs, IEEE trans. Software Eng S-5, 1978, pp 440-452.

BRYANT, R.E. Simulation of Packet Communication Architecture Computer Systems, 1977, Computer Science Laboratory, M.I.T, Cambridge.

JEFFERSON, D AND SOWIZRAL, H. Fast concurrent simulation using the Time warp mechanism, part I: Local Control, Technical report N-1906-AF, RAND Corporation, December 1982.