# Towards Adaptive Caching for Parallel and Discrete Event Simulation

Abhishek Chugh and Maria Hybinette

Computer Science Department
The University of Georgia
415 Boyd Graduate Studies Research Center
Athens, GA 30602-7404, U.S.A.

maria@cs.uga.edu

## 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.

## 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 (LP) 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 section covers related work in this area. Our caching approach is described in section 3 . The implementation and the programming interface is described in section . Section 5 discusses performance results. Advantages and limitations are outlined in section 6 and we discuss future work in section 7. The paper closes with a conclusion and discussion.

## 2 Related Work

Different techniques for reusing computations have been proposed and implemented earlier. 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 mechanism are appropriate for multiple *similar* simulation runs.

Memoization or function caching is a technique where inputs and the functions corresponding results are cached for later re-use. This technique has been around for over 40 years (Bellman 1957; Michie 1968). Functional caching is widely used for incremental computations, dynamic programming, and many others. In particular, incremental computation is a technique that takes advantage of repeated computations on inputs that differ slightly. It makes use of previously computed results in computing a new output. Using functional caching to obtain efficient incremental evaluation is discussed in (Pugh and Teitelbaum 1989). Deriving incremental programs and caching intermediate results provides framework for program improvement (Liu and Teitelbaum 1995).

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 prior knowledge of computation is required to break it into sub-computations. 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).

Our approach is applicable both to optimistic and conservative protocols and is an adaptive approach that considers 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, however, it may complement the approaches above such as staging or cloning. This is different than the studies reported above.

## 3 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 1). 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.

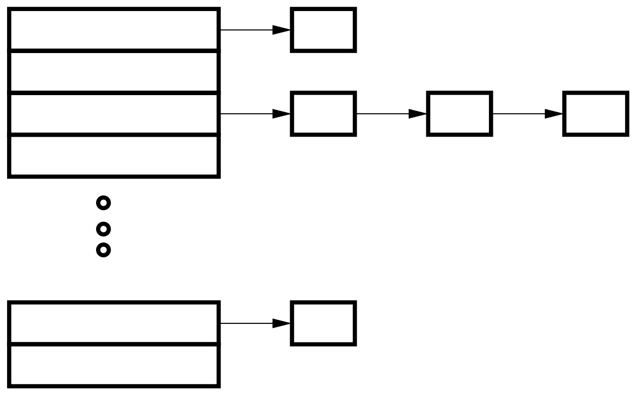

Figure 1: Caching Implementation: The cache is implemented as an array that is indexed as a hash function. The hash table uses separate chaining to resolve collisions and is implemented as an array of linked lists.

### 3.1 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.

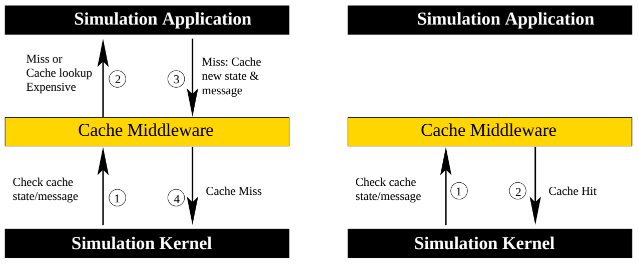

Figure 2: Caching Middleware: Our caching scheme is implemented as middleware between the simulation application and the simulation kernel. The left diagram illustrates the sequence of events in the case of a cache miss. The diagram on the right shows a cache hit.

We provide an API for the user application to the middleware. Figure 2 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.

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.

### 3.2 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 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.

## 4 Cache Implementation

### 4.1 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 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 $\frac{1}{4}$ 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.

## 4.2 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 take longer to process the event than to reference the cache (times some multiple) we reference the cache.

## 4.3 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 wrap-up. These phases are described in more detail below.

## 4.4 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 );
  }
```

## 4.5 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()`

### 4.6 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 1 on success and 0 otherwise. It does not take any input argument.

## 5 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 the P-Hold application on GTW, an optimistic time warp simulation kernel (Fujimoto 1990; Das et al. 1994). 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.

### 5.1 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. The plots in Figure 3 show that hit ratio generally increases as we

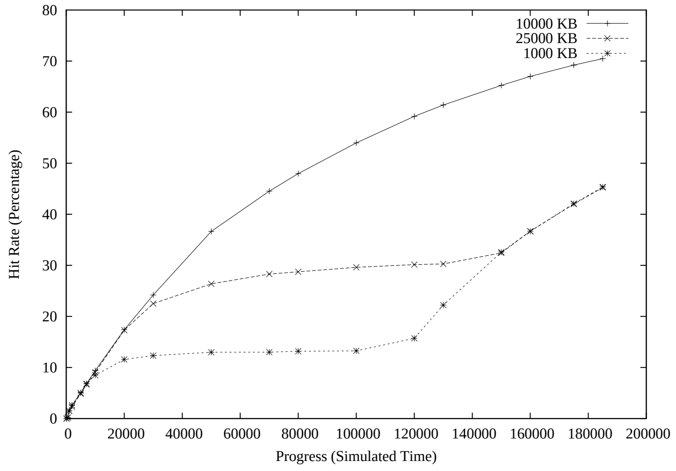

Figure 3: Performance of P-Hold: The hit ratio increases increases as the simulation progresses. Larger numbers indicate better performance.

increase the length (number of events) of the simulation. Three experiments were run, using different cache sizes: a) cache size same as the size required to store all results, b) cache size one-fourth the size required to cache all results, and c) cache size one-tenth of the size required to cache all results. 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 $\frac{1}{2}$ 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 P-Hold application our hit ratio approaches 70%, thus forcing 30% of the computations, leading to a speedup limit of about 3.3.

## 5.2 Basic Caching Experiments: Speedup

We evaluated speedup in comparison to traditional simulation (without caching) with respect to: the size of the cache and the running time of the simulation (proportional to the number of events processed).

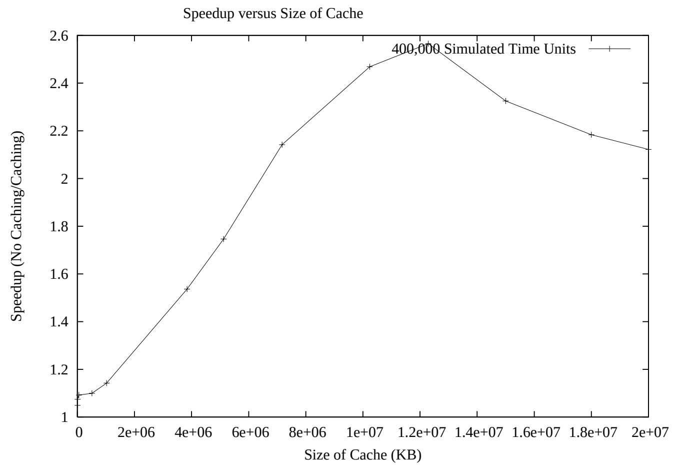

Figure 4: Performance of P-Hold: Speedup in comparison to traditional simulation (without caching) with respect to: the size of the cache and the running time of the simulation. Larger numbers indicate better performance.

Figure 4 shows speedup versus the size of the cache for different simulation running times. Speedup improves as the size of the cache is increased. We see the best speedup in the case of the longest running time. This is because we are allowed to run for a longer time after the warm up phase. However, beyond a certain point as cache size is increased, speedup declines, then levels off.

We suspect that the drop in speedup corresponds to the cache size at which the memory requirements for the simulation exceed the physical size of memory alloted to the process. At this point, the underlying OS relies on virtual memory techniques to provide the resources. Performance does not continue to degrade because the caching software is allowed to maintain a useful working set in RAM. In other conditions, for instance when physical memory is small in relation to effective cache size we may see a more aggressive drop in performance as virtual memory is utilized.

## 5.3 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 10 elements. This change forces a linear search for matches much more often. While a hash table of 10 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.

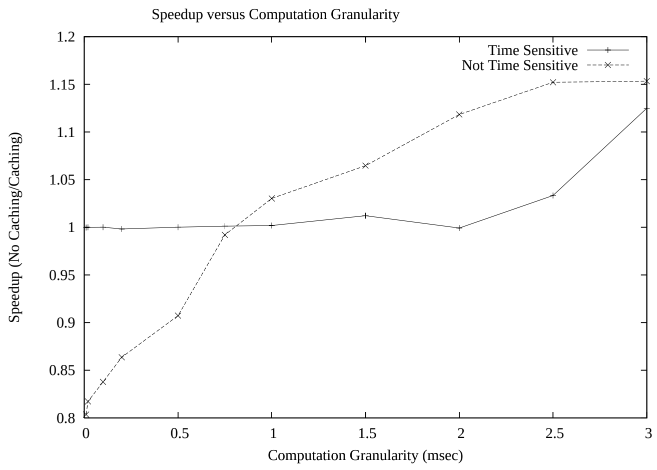

Figure 5: Performance of P-Hold: Shows speedup results for simulations using simple caching and adaptive time sensitive caching. Larger numbers indicate better performance.

Figure 5 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 four 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 1.0 milliseconds. This means that the adaptive technique improves performance over simple caching in this region.

As event granularity increases beyond 1.5 milliseconds, both simple caching and adaptive caching begin to exceed speedups of 1.1. In future work we will continue to evaluate the factors affecting performance in this region.

## 6 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.

## 7 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 re-evaluating *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).

## 8 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* 16$^{th}$ *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 SIGACT-SIGPLAN: ACM Press.

WALSH, K. AND SIRER, E. G. 2003. Staged simulation for improving scale and performance of wireless network simulations (Dec. 2003).

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