Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. Hill, L. Moench, and O. Rose, eds.

TRANSPARENT AND ADAPTIVE COMPUTATION-BLOCK CACHING FOR AGENT-BASED SIMULATION ON A PDES CORE

Yin Xiong, Maria Hybinette and Eileen Kraemer

Computer Science Department
University of Georgia
Athens, GA 30602-7404, USA

ABSTRACT

We present adaptive computation-block caching that supports improved performance and is suited for agent-based simulations. The approach is illustrated in SASSY (Scalable Agents Simulation System). SASSY leverages a Parallel Discrete Event Simulation for performance, but provides an agent-based API to the developer. Agent-based simulation is suited to computation-block caching because relevant calculations completed at each event may be relatively heavyweight and may be repeated. The potential savings of avoiding a computation entirely may offset the overhead cost of caching. The approach is refined through the use of statistical methods for choosing which computation blocks should be cached or not. If the relevant computation is trivial, caching is not worth the cost. In other cases caching provides a substantial speedup. Our mechanism tracks these costs online and adjusts accordingly. It requires no additional coding but is automatically integrated into applications. We assess the performance of the approach in a benchmark-application.

1 INTRODUCTION

Caching the results of expensive and redundant computations or database retrievals improves application scalability and execution time. The idea of providing caching is not new but has been around since the 1960s when it was first introduced to improve the performance of the Model 85, part of the System/360 IBM product line. Typical parallel and distributed discrete event simulations (PDES) recompute events in time stamp order, without exploiting a computational result cache even if events may have been processed earlier. It is thought that for most such simulations events are fine grained (light weight) computations and these computations do not offset the overhead of caching enough to provide an improvement in performance. However a growing need exists for simulations that support agent-based simulation, in which events are coarser-grained than the events assumed by traditional PDES systems.

PDES events typically require less than a millisecond (Steinman and Wong 2003, Das et. al. 1994), while agent-based simulation events can run for tens of milliseconds. Agent based simulations of robots (e.g., TeamBots (Balch 1998) and Player/Stage (Gerkey et. al. 2003)) often assume a time step rate of 33 msec as this corresponds to the frequency at which a video camera delivers images. Further, all of the intervening time is typically used to process the information and compute a movement command. However, these agent-based simulations do not scale well.

Agents in an agent-based simulation (ABS) normally rely on a *sense-think-act* cycle. Agents sense the environment, consider what to do, and then act. Tile World, a test bed to evaluate reasoning of agents, requires substantial time to deliberate (Pollack and Ringuette 1990). Tile World was proposed in 1990 by Pollack and Ringuette and consists of a grid of cells on which various objects, such as tile workers, tiles, obstacles and holes, can exist. The tile workers (agents) can move up, down, left or right, and their objective is to pick up and move tiles so as to fill holes. Each hole has an associated point value that is awarded to the agent upon filling the hole. A hole varies in size and point value. The agents know how valuable each hole is in advance; their overall goal is to get as many points as possible. Tile world simulations are dynamic because the environment changes continually over time. The objects appear and disappear at rates pre-determined by parameters of the simulator.

(Uhrmacher et. al. 2000) implemented Tile World in JAMES, a DEVS based simulation system, and found that "thinking time" required almost 80% of an agent's time step, where a time step was close to 1 second. The sense and act components used less than 20% of the total simulation time. (Lees et. al. 2004) parameterized thinking time and compared thinking and reactive agents (where reactive agents require little or no thinking time as they react directly to their sensor inputs) in a shared and central environment in Tile World. They experimented with a modest

number of agents (up to 64 agents) using a Linux cluster. The deliberating agents use an A* planner to generate plans of routes to tiles and holes within the tile world. Their planner incorporates a 10 ms "deliberation delay" per plan. A* is a classic and frequently used planning algorithm in agent based simulations that finds the least cost path between an initial point and a goal. It was proposed by (Hart et. al. 1968) in the late sixties. A* provides an optimal solution plan to the path planning problem but it does not provide any performance guarantee. A* overhead ranges between 10 ms and 1,000 ms on a 2GHz Pentium (Balch 2008). While there are many extensions to A* (e.g., D* uses the initial plan as a baseline to plan new paths in dynamic environments instead of recreating the path from scratch (Stentz, A. 1994)) and alternate planning algorithms, A* remains popular as it is simple to implement and provides descent performance.

The thinking step independent of particular planning algorithm, as observed by the agent-based simulation community, ranges from a complex step requiring lengthy computation intensive to a reactive step with negligible 'thinking time'. Accordingly, the performance of an agent-based simulation can be improved significantly by speeding up the thinking process. We exploit variable thinking time and use *adaptive* caching in which we cache the input parameters and the results of lengthy thinking in order to avoid re-computation – but avoid caching computations where the relevant time is trivial, such as with reactive agent that do not think, where caching may not be worth the cost.

An agent's thinking process may involve several input parameters and possibly depend on a large state space, and the probability of encountering exactly the same set of parameters and state variables can be low. Thus, caching the ultimate result of the whole thinking process may not be beneficial as the cache hit rate can be minimal. Here, our approach of block caching enables breaking the thinking process into smaller units that may be more amenable to a caching mechanism and less (as a whole) dependent on the state space.

Our caching is flexible and transparent to the user, the application developer, as it requires no additional coding or recoding. By using a software cache pre-processor, caching code is integrated and compiled automatically and transparently with the developer agent applications. Currently we run the processor before starting the simulation but in theory it can be run dynamically, on-the-fly, while the simulation is running. Our motivation is to make caching transparent to the user while improving scalability and performance.

2 RELATED WORK

Caching is used in different applications and is integrated at different levels into the architecture including software, language systems and hardware. Function caching or memoization is a technique suggested by the programming language research community to improve the performance of functions by avoiding redundant computations. Here, function inputs and corresponding results are cached in anticipation of later reuse (Bellman 1957; Michie 1968; Pugh 1989).

Function caching is used for incremental computations, dynamic programming and in many other situations. Incremental computations allow for slight variation in function input. It makes use of previous results and adjusts it to generate new output. Using function caching to obtain efficient incremental evaluation is discussed in (Pugh and Teitelbaum 1989). Deriving incremental programs and caching intermediate results provides a framework for program improvement (Liu and Teitelbaum 1995). Memoization is available today as part of the Java programming language.

Walsh and Sirer proposed *simulation staging*, a form of function caching, as a way to improve the performance of a sequential discrete event simulation in applications with a substantial number of redundant computations (Walsh and Sirer 2003). Their approach provides significant speedup (up to 40x in a network application), but requires extensive structural revision of code at the user application level.

Contrary to our approach, function caching techniques do not consider the cost of consulting the cache and are not adaptive. Observe that if the cost of checking the cache exceeds the cost of just doing the computation, caching will degrade performance. Function caching also relies on an assumption of no side effects (e.g., by variables in the state space) and that the function produces only one output.

The PDES community have proposed different techniques of reusing computations. 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 and Fujimoto 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 used in optimistic simulators to improve the performance of rollbacks, lazy re-evaluation, caches the original event in anticipation of it being re-used after a rollback and thus avoiding re-computation (West 1988). Lazy evaluation, however, is only beneficial for events on the same execution path.

We recently developed *LP caching* (Chugh and Hybinette 2004) for parallel and distributed simulators. LP caching is distinct from the work presented in this paper, *block caching*. Both approaches are independent of the simulation engine (i.e., it supports both conservative and optimistic simulation executives). However, in LP caching the middleware exploits the PDES *paradigm* of logical processes (LPs) and messages by intercepting communications between the simulation application and the simulation executive (See left of Figure 1). Here the caching middleware is situated between the simulation kernel and the simulation application. When the kernel delivers an event to the kernel, the caching software intercepts it. In the case of a cache hit, the retrieved resultant state and message or messages are passed back to the kernel without the need to consult the application code. This scheme, as with our proposed approach, is also adaptive in the sense that it avoids consulting the cache when the computation is negligible. A significant between LP caching and block caching is that block caching does not rely on a simulation paradigm but can be plugged in to a variety of applications and application levels (see left of Figure 1 for an example on how it is be integrated with a PDES simulation); it is simulation independent. Block caching can improve the performance of functions or blocks transparently without any need for application developer intervention (however a block or chunk of code currently requires annotations at the beginning and end of potential block of code with a comment) of both the simulation application and simulation executive. Similar to (Walsh and Sirer 2003)'s approach (it can split a large computation into smaller sub-computations whose inputs and result(s) are cached to further improve performance.

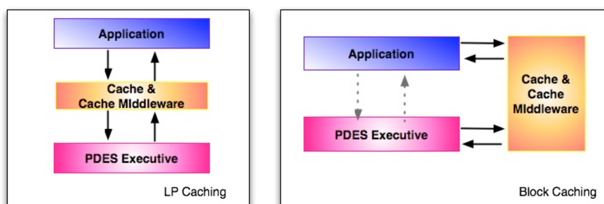

Figure 1: Caching Approaches: Our earlier approach (left) and our proposed approach right.

Our goal of transparency is inspired by JiST, which infuses sequential discrete simulation semantics directly into the Java Virtual Machine (JVM) to provide a transparent user programmer interface (Barr et. al. 2004). In JiST a rewriter reprocesses or rewrites simulation application class code in order to incorporate embedded simulation time operations. The rewriter is a dynamic class loader. It intercepts all class load requests and subsequently verifies and modifies the requested classes. The program transformations occur once, at load time, and does not rewrite the during execution. Although JiST does not provide caching functionality we hope in future work to explore embedding our caching middle-ware into the JVM to improve the interface and further transparency.

We propose *computation-block caching*, a transparent, flexible and adaptive approach to reduce redundant computations. It is transparent in the sense that no recoding is required on the part of application programmers. It is flexible since it can decomposable large computations into smaller and potentially re-order to improve performance. It is adaptive in the sense that the caching mechanism is turned on when statistics shows that the benefit of caching exceeds computation by a pre-specified factor. In the next sections we will discuss the approach, implementation and discuss initial performance results.

3 APPROACH

We define *computation block* to mean a chunk of code that may be a Java method or a number of lines of code with or without invocations of methods. Computation-block caching is not as rigid as traditional function caching. It allows state variables to be involved in caching and the result it returns is not limited to returning a single value. Consider the following computation block as an example:

```
int a;
int b;
methodA( a, b, c, d );
if( c > d )     // c, d: state variables)
     doSomething( c );
else
     doSomethingElse( d );
```

For traditional function caching, this chunk of code is not easily cacheable because it violates the basic rules for function caching, namely, it is not a function, but involves multiple functions and state variables. But the simulation application may have every reason to want to cache this chunk of code. One way for traditional function caching to solve the problem is to cache the functions separately, but the amount of recoding will be substantial as each function will need some recoding in order to make it cacheable. Furthermore the functions may write or read from variables that are not passed in as parameters (e.g., variables a and b). The state variables that affect the functions need to be denoted and their updated values need to be copied back to the state variables.

Block caching solves the tedious task of recoding by utilizing a preprocessor that automates the process by *generating* a new version of the code, on-the-fly, that includes calls to the caching middleware. To designate a computation function as "cacheable", the application programmer provides a method-signature specification in a configuration file. The following is a sample specification for a cacheable computation function or method called `dummy1` in the original code:

```
    begin:dummy1
    packageName: app
    className: JPhold
    return: length=double, point=int
    parameters: int a, double b
    stateVariables: int sex, int age
    cachingFlag: on
  end:dummy1
```

Here "`dummy1`" is the name for a cacheable function. "`JPhold`" is the name of the Java class containing the function. "`app`" is the name of the Java package that "`JPhold`" belongs to. The passed-in parameters are an "`int`" and a "`double`". There are two state variables involved in the computation: "`sex`" and "`age`". The result to be cached is the value of variable "`length`" whose data type is "`double`" and of variable "`point`" whose data type is "`int`". The caching flag for this function is set to be "`on`" for this particular run.

For a computation block that is not a Java method, but a chunk of code, we require that the application programmer mark the beginning and end of the block in their Java class code. Note that this is not "recoding" as the markers are Java *comments* and they do not change the byte generated code. Taking the above computation block as an example (which is a chunk of code rather than a function), the modified class code would look like this:

```
//beginComputationBlock dummy2
int a;
int b;
methodA(a, b, c, d);
if (c > d)//c,d: state variables)
     doSomething(c);
else
     doSomethingelse(d);
//endComputationBlock dummy2
```

A simulation application can designate multiple computation blocks as "cacheable". A cacheable computation block does not need to be cached all the time. The user can specify which computation blocks to be cached for a certain simulation run by turning on the caching flags in the specification file. The caching flags can be set before the simulation begins to run and remain unchanged throughout the simulation, which is called "hard-caching". The caching flags can also be set on or off during the simulation run according to statistics computed on-the-fly, which is called "soft caching".

3.1 The Caching Middleware

Our implementation includes two modules: a preprocessor that reads a configuration file and generates code on the fly and the cache middleware that manages caching and determines whether to consult the cache or not. Figure 2 depicts the interactions between the caching modules and a pre-existing PDES simulation executive and its simulation application. The pre-processor first reads a configuration file or stream (a stream if it generates code while the simulation is running) then 'recompiles' the effected objects (red dashed arrows in the Figure denotes the flow of output of code to the effected modules).

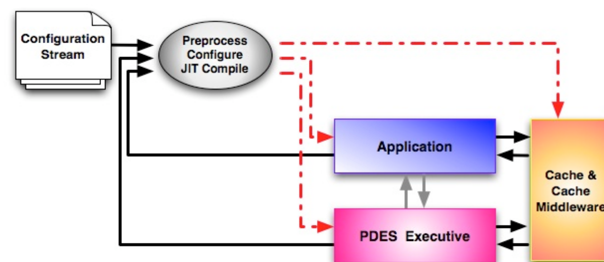

Figure 2: Workflow of Preprocessing

The regenerated code enables the cache middleware to intercept and monitor cacheable function calls (or blocks) in both the simulation kernel and the simulation application. The cache is consulted when the overhead of the computation time exceeds the caching overhead.

To provide user control whether functions or blocks are cached – a caching flag can set be set an unset on a per block basis. A block's flag can be changed at any time, before the application runs or while it is running. The state of the flag (on or off) is set in the configuration stream. A Statistic Manager (part of the cache middleware) keep track of cache and computational overhead to determine the threshold when to consult the cache or not. When the Statistic Manager determines it is worthwhile to consult the cache and it is a hit it returns the cached results. In the case of a cache miss, the cacheable computation block is carried out and the result is cached for later reuse.

We implemented the cache middleware to run both in distributed mode across several machines or on a single machine. Both version can build multiple caches on a single machine.

The cache is implemented as a Java HashTable and indexed by the combination of package name, class name, computation name, passed-in parameters and the names of state variables involved in the computation. The result of the computation is stored with the index as a key-value pair in the hash table. Our caching middleware can be used with both conservative and optimistic simulation kernels (or any application). It can also be used with both ABS simulation and non-ABS simulations. No changes to the underlying kernel are required. No changes to the simulation application are required.

3.2 The Preprocessor

Existing caching schemes are not suitable for our purposes because they usually require substantial recoding in order to use the caching facilities. By "recoding" we mean manually modifying the code of the cacheable functions,

such as adding, deleting or rewriting lines of code. Therefore, such caching schemes involve "hard coding" which can be error-prone and time consuming. For cacheable functions, the recoding is usually on a function-by-function basis, *i.e.*, for each cacheable function, the application programmer needs to do some recoding in order to make that function cacheable. For example, in (Chugh 2004), a cacheable function needs at least 4 lines of recoding. For LP caching, however, a 4-line recoding may not be too much as it caches one cacheable function per LP. But for computation-block caching, LP events be decomposed and into multiple computation blocks to be "cacheable" (note that decomposing a function may also relieve chunks of code (or functions) to be dependent on less state variables and each other if reordering is advantageous). If each computation block needs 4 lines of recoding to make it cacheable, the amount of recoding necessary may make the task intimidating and time consuming.

The preprocessor in block caching completely relieves the application programmer of recoding in order to make a computation block cacheable. As Figure 2 shows, the Preprocessor reads the configuration file and involved application Java files to generate a new version of the application Java files, inserting caching-specific code that checks whether the caching flag is on and accesses the cache if necessary. Also Cache middleware is updated accordingly. There is no need to invoke the Preprocessor for each simulation run. It is invoked only when the specification for the cacheable computations is modified.

The time for preprocessing is decided by a few parameters: the number of cacheable computation blocks, the number of class files, and the length of class files. The Preprocessor scans the configuration file to find which application Java files are involved in caching, then reads the files one by one and inserts caching-specific code at the right places.

3.3 The Statistics Manager

A feature of our method is that it allows both "hard caching" and "soft caching" options (recall that soft caching enables adaptive caching). The Statistics Managers manages soft and hard caching. The Statistics Manager is composed of two sub-managers. One sub-manager computes the average caching overhead and the cost for each cacheable computation block on the target computer system. Users run this sub-manager on their system and then compare the computation cost with the caching overhead to decide whether the "caching" flag should be turned on, and if on, what threshold value should be selected. The other sub-manager gathers information about the parameters, state variables and length of the computation as the simulation runs. It then decides whether the caching flag should be turned on or off for a certain cacheable computation. If the benefit of caching surpasses a certain threshold specified by the user beforehand, or generated on-the-fly, caching will be turned on, otherwise, it will be turned off.

4 PERFORMANCE

Caching efficiency depends on at least three factors: cost of a cacheable computation, number of such computations, and the caching overhead. In general, we expect better performance from caching as the cost of computation increases and as the cost of cache consultation decreases. 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, the performance improves. Accordingly, longer simulations are more likely to benefit from caching. The size of the cache is also important because for a given cache size, the number of key-value pairs stored is inversely proportional to the size of the cache. When the number of key-value pairs exceeds the cache size, either some of them will be cleared from the cache, or the cache size has to be increased, which means allocation of new memory space and a large amount of copying.

In our experiment, quantitative results were obtained using JPHold, a Java version of the PHold application (Fujimoto, 2001). JPHold provides a synthetic workload using a fixed message population. Upon receiving a message, the LP schedules a new event whose destination LP is drawn from a uniform distribution ranging from 0 to one less than the number of LPs, which means that each LP is equally likely to be the destination of a message.

We tested our caching scheme on SASSY, an optimistic PDES simulation executive implemented in Java (Hybinette et. al. 2006) running of UNIX Workstations (primarily SUN Ultra workstations) connected via Ethernet/Fast Ethernet to SUN Microsystems. Three types of experiments were performed: 1) Experiments as proof of concept of the basic caching technique; 2) Experiments to evaluate the role that pre-run statistics play in aiding decision making; and 3) Experiments to study the benefit of adaptive caching using statistics computed on-the fly.

Each of our experimental runs is defined by a set of parameters: the number of PEs (simulation schedulers), the number of Logical Processes (LPs, an LP is logically a sequential simulation 'process' scheduled by a PE), the message population, total events to be processed, the initial cache size, the load factor of the hash table, the computation granularity and more. For our experiments reported here, we used 10 machines that ran 40 PEs with a total of 1000 LPs evenly distributed over the 40 PEs. As workstations may have external loads and processes (not necessarily related to our simulation runs) while we ran our experiments we averaged the run time over all LPs to get the "mean time per event" which is then used in the speedup computation. For each setting, we ran the simulation 10 times and used the mean time in our reported results.

4.1 Basic Caching Experiments: Speedup

We evaluated speedup by running the same program with a certain setting for both cache-on and cache-off options. That is, for each run, we controlled for all parameters except the cache-on/cache-off parameter. No overhead is introduced in the cache-off condition.

Figure 3 shows the speedup of cache-on vs. cache-off by computation granularity. We defined granularity according to the caching overhead and computation cost on our system. The lowest line represents the speedup for computations with fine granularity, which had a mean processing time of 1.689 ms; the middle line represents the speedup for computations with mid granularity, which had a mean processing time of 6.498 ms; and the top line represents the speedup for computations with coarse granularity and a mean processing time of 16.053 ms.

From Figure 3 we can see that the speedup is dominated by computation granularity and number of total events processed. Coarse granularity computation resulted in the greatest speedup. For the same granularity, the longer the computation runs, the greater speedup we would gain by turning cache on, until the cache hit rate approaches 100%, at which point the speedup curve flattens out.

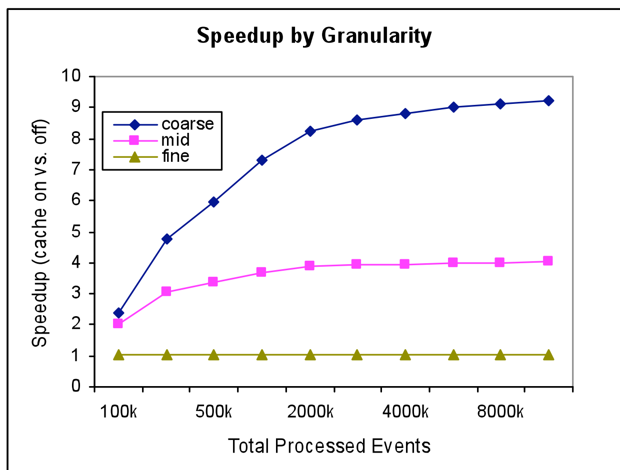

Figure 3: Speedup by Computation Granularity

4.2 Pre-run Statistics Computation

To get an idea of the overhead of caching the Statistics Manager to collected and computed statistics about the cache overhead and the cost of the cacheable computations on our workstations using a benchmark application. The benchmark uses a Fibonacci computation to measure the caching overhead. The Fibonacci sequence has some qualities that suit measuring the overhead. First, it needs only 1 parameter so we can easily control the range of this parameter which, in turn, controls the cache hit rate; second, the time needed for the recursive computation of Fibonacci covers a wide spectrum of time lengths, so we can generate workload of all kinds of granularities with the Fibonacci function; and third, it is easy to implement. Of-course the caching overhead varies depending on the application but a Fibonacci benchmark provides a reasonable ballpark estimate.

The mean computation time for different values of the input parameter *k* by running Fibonacci on a certain *k* 100 times and then computing and recording the average time. Table 1 shows a few lines from the statistics we gathered about running Fibonacci on our system:

Table 1: Computation Costs

k	result	mean	Cumulative mean
20	6765	0.16	0.025
30	832040	19.31	1,689
31	1346269	31.31	2.644
35	9227465	214.9	16.05
40	1.02E+08	2379.3	155.7

In the above table, *k* is the input parameter for the Fibonacci function. The “result” column contains the result for the Fibonacci function with input *k*. The “mean” column shows the mean cost for computing Fibonacci numbers with a certain *k*. The last column contains the cumulative mean, which is the mean for the computation costs of Fibonacci sequence with parameters from 1 to *k*, namely, the mean of Fibonacci(1) + Fibonacci(2) + … + Fibonacci(k). The Fibonacci benchmark indicated that it is worthwhile to cache a function (or block) on SASSY when the granularity of computation is at least 1.5 ms (this is for 10 machines and the test environment described earlier).

4.3 Adaptive Caching Experiment: Hard Caching

With the pre-computed statistics presented in the previous section, we know that any computation with a granularity greater than 1.5 ms is a potential candidate for our caching scheme, i.e., turning on cache will potentially enhance performance. As a test, we selected a computation block which has a computation granularity of 2.64 ms. We designated it “cacheable” and turned on the caching flag for this computation block.

Figure 4 shows the speedup of cache-on over cache-off for this computation block.

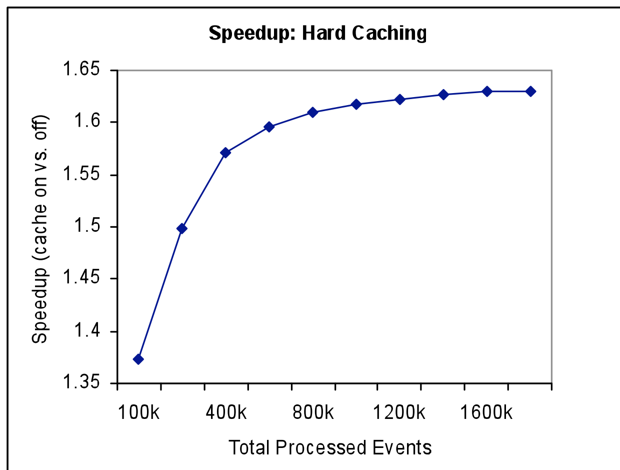

Figure 4: Speedup: Hard Caching

4.4 Adaptive Caching Experiments: Soft Caching

Relying on pre-computed statistics is appealing because it is easy to use and the performance enhancement is guaranteed if the computation granularity can be accurately computed beforehand. For simple computations, especially those driven by random numbers, if we know the distribution of the random numbers, we can use our Statistics Manager to obtain computation granularities in advance. But for computations that involve parameters whose distributions are unknown beforehand, it is hard to compute statistics for their computation granularities without running the simulation.

Our Statistics Manager continuously collects statistics while the simulation is running — so it is not necessary to provide pre-computed statistics. It computes (and re-computes) statistics on-the-fly and makes decisions as to whether the cache should be turned on or off for a certain cacheable computation, according to the statistics and a threshold (pre-computed or not).

To test the on-the-fly decision making effectiveness of the Statistics Manager, we modified the previously-mentioned computation block to involve one state variable in its computation. The state variable is "energy" which indicates how much energy the agent possesses, which helps the agent to decide whether the task is worth taking up. If "energy" is lower than a predefined threshold, the agent gives up the task until a later time when its "energy" is regained. We then ran the testing program with cache-off , "hard caching" and "soft caching" options. With "soft caching", the Statistics Manager starts by gathering information about the cost of the computations and frequency of cache reference. After some time, it accumulates enough information to approximate the cost for the passed-in parameters and the state variable. When it sees those parameters and the state variable, it first finds out the approximate cost and compares the cost with the pre-computed cache overhead. If the cost is greater than the threshold, it turns cache on. If the cost is less than the threshold, it turns the cache off.

Figure 5 shows the speedup of "hard caching" vs. "soft caching". The blue (lower) line represents the speed up gained over cache-off by "hard caching", i.e. cache is turned on at the very beginning of the simulation (and does not change). The pink (top) line represents "soft caching", i.e., the cache is turned off for fine-granularity computations and on for coarse-granularity computations.

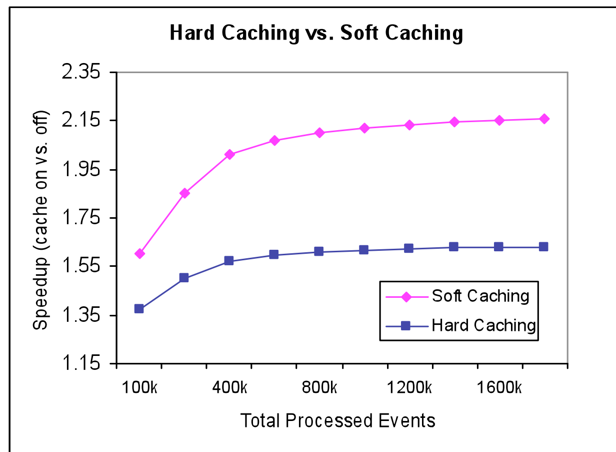

Figure 5: Hard Caching vs. Soft Caching over Cache-off

"Hard caching" and "soft caching" have their own favorite cases where one performs better than the other. For those computation blocks that mainly rely on input parameters whose distribution can be decided in advance, "hard caching" is more advantageous because by the help of the Statistics Manager we can easily find out its computation cost. But for computation blocks that involve parameters whose distribution relies on run-time situation, "soft caching" would be more advantageous because the Statistics Manager will "learn" from the changing situation.

5 CONCLUSIONS AND FUTURE WORK

We designed and implemented *computation-block caching*, a new caching scheme, and experimentally proved its merits in applicability and performance.

The proposed caching mechanism handles both side effects (or dependencies of state variables) and the return of multiple results. The computation blocks are not limited to functions (or methods). It does not require recoding either on the application level or on the kernel level. We designed and developed a preprocessor that reads the application-provided specifications and generates a cacheable version for each specified computation block. The specification for cacheable computation blocks can be modified

any time as needed. The preprocessor is invoked only when modifications are made to the specifications.

Further the caching scheme is adaptive in the sense that cache can be turned on and off for each individual cacheable computation block according to statistics gathered before hand or on-the-fly. We provided a Statistics Manger to facilitate both hard caching and soft caching.

We experimentally proved that caching performance is dominated by computation granularity while also affected by many other factors including cache hit rate and length of simulation, all of which can be manipulated to improve performance.

We have tested our caching scheme on the JPHold and Fibonacci benchmark programs. Our next step is to test our caching scheme on standard ABS simulation applications such as Tile World and soccer simulations.

REFERENCES

Balch, T. 1998. Behavioral diversity in learning robot-teams. Ph. D. thesis, College of Computing, Georgia Institute of Technology.

Balch, T. 2008. Personal Communication, College of Computing, Georgia Institute of Technology.

Barr, R., J. Zygmunt , R. R. Haas. 2004. JiST: Embedding Simulation Time into a Virtual Machine. *Proceedings of EuroSim Congress on Modeling and Simulation* September 2004.

Chugh A. and M. Hybinette. 2004. Towards Adaptive Caching for Parallel and Discrete Event *Simulation. Proceedings of Winter Simulation Conference* 2004: 336-343.

Ferenci, S., R. M. Fujimoto, M. H. Ammar, K. Perumalla and G.R. Riley. 2002. Updateable Simulation of Communication Networks. In *Proceedings of the Workshop on Parallel and Distributed Simulation* :107-114.

Fujimoto, R. M. 1990. Performance of Time Warp under synthetic workloads. *Proceedings of the SCS Multi-conference on Distributed Simulation* Volume 22: 23–28.

Gerkey, B., R.T. Vaughan, and A. Howard, 2003. The-Player/Stage project: Tools for multi-robot and distributed sensor systems. *Proceedings of the International Conference on Advanced Robotics*, 317–323. Coimbra, Portugal.

Hybinette, M., E. Kraemer, Y. Xiong, G. Matthews and J. Ahmed. 2006. SASSY: A Design for a Scalable Agent-based Simulation System Using a Distributed Discrete Event Infrastructure. *Proceedings of the 38th conference on Winter simulation.*

Lees, M. 2002. A history of the Tileworld agent testbed. *Computer Science Technical Report No. NOTTCS-WP-2002-1*. http://www.cs.nott.ac.uk/WP/2002/2002-1.pdf [accessed March 29, 2008]

Lees, M., B. Logan, T. Oguara, and G. Theodoropoulos (2004). "HLA_AGENT: Distributed Simulation of Agent-Based Systems with HLA." Proceedings of the International Conference on Computational Science (ICCS'04) (pp. 907-915).

Liu, Y. and T. Teitelbaum. 1995. Caching Intermediate Results for Program Improvement. *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (La Jolla, CA, June 1995), 190–201.

Logan, B., Theodoropoulos, G. 2001. The Distributed Simulation of Agent-Based Systems. *IEEE Proceedings Journal, Special Issue on Agent-Oriented Software Approaches in Distributed Modeling and Simulation*, February 2001.

Hart, P. E., Nilsson, N. J.; Raphael, B. 1968. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics* SSC4 (2): pp. 100–107.

Pollack, M. E., and M. Ringuette. 1990. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI Press, pp. 183-189.

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

Pugh, W. and T. Teitelbaum. 1989. Incremental computation via function caching. *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages* (Austin, Texas, Jan. 11–13, 1989), 315–328.

Steinman, J. S. and J. W. Wong. 2003. The SPEEDES persistence framework and the standard simulation architecture. *Parallel and Distributed Simulation, 2003 Proceedings.* Volume 10-13 (June 2003): 11 – 20.

Stentz, A. 1994. Optimal and efficient path planning for partially-known environments. In *IEEE International Conference on Robotics and Automation*.

Uhrmacher, A. M., P. Tyschler and D. Tyschler. 2000. Modeling and simulation of mobile agents. *Future Generation Computer Systems* 17 (2): 107–118.

Walsh, K. and E. G. Sirer. 2003. Staged simulation for improving scale and performance of wireless network simulations. *Proceedings of Winter Simulation Conference, 2003.* Volume 7-10: 667 – 675.

Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. Hill, L. Moench, and O. Rose, eds.