Staged Simulation: A General Technique for Improving Simulation Scale and Performance

KEVIN WALSH and EMIN GÜN SIRER
Cornell University

This paper describes *staged simulation*, a technique for improving the run time performance and scale of discrete event simulators. Typical network simulations are limited in speed and scale due to redundant computations, both within a single simulation run and between successive runs. Staged simulation proposes to restructure discrete event simulators to operate in stages that precompute, cache, and reuse partial results to drastically reduce redundant computation within and across simulations. We present a general and flexible framework for staging, and identify the advantages and trade-offs of its application to wireless network simulations, a particularly challenging simulation domain. Experience with applying staged simulation to the **ns2** simulator shows that staging can improve execution time by an order of magnitude or more and enable the simulation of wireless networks with tens of thousands of nodes.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*; I.6.7 [**Simulation**]: Simulation Support Systems

General Terms: Wireless Simulation Performance Scale

Additional Key Words and Phrases: Staged simulation

1. INTRODUCTION

Network simulations are critical for the design and evaluation of distributed systems and networking protocols. Especially in newly emerging domains such as mobile ad hoc and sensor networks, simulations are crucial to evaluating new systems and protocols across a range of deployment scenarios. While thorough evaluations require accurate, efficient and scalable network simulators, achieving all three of these properties simultaneously is a challenging task. We assert that a significant source of inefficiency in discrete event simulators stems from redundant computation, and that this wasted computation presents a significant limit to simulation speed and scale. We identify two different classes of redundancy in traditional discrete-event network simulators.

The first class of redundant computation occurs within a single run of the simulator, and stems from an overly strict notion of accuracy. Traditional network simulators are conservative, that is, they reevaluate complex functions whenever their results *may* have changed, even though in reality the results may have changed very little, if at all, since the last time they were evaluated. An illustrative example is the computation of a node's one hop neighbor-set in wireless network simulations. This is an expensive and frequently-used primitive in wireless simulations. It computes the positions of all nodes relative to the sending node, calculates the power level of the signal at each receiver, and determines the set of nodes that can capture transmitted packets. In the most general case, nodes may change position, alter sending and receiving power thresholds, or modify antenna models or parameters at any time. A conservative wireless network simulator which has computed the neighbors of a wireless node at time t will still recompute the neighbor-set from scratch at time step $t+\varepsilon$. In reality, however, few wireless simulation scenarios will change dramatically over short time-scales. Nodes move smoothly and relatively slowly, if at all. Other variables, such as antenna model, reception threshold, and transmission power change infrequently. In general, the neighbor-set computed at time t is likely to remain valid for some time, changing slowly and gradually as nodes move into or out of range. Much of the time spent re-computing neighbor-sets from scratch at each time step is therefore effectively wasted.

A second class of redundant computation occurs between multiple runs of a simulator, and stems from performing calculations which were already computed in previous runs. Simulations are often performed in batches of tens or hundreds of simulation runs, with only slightly varying simulation parameters within each batch. While each run will differ in some inputs, such as the random number seed, network traffic pattern, or protocol parameters, often there is still a significant overlap between the work performed in multiple invocations of the simulator. Executing each simulation independently and without the benefit of past computations leads to redundantly computing many identical results.

This paper describes *staged simulation*, a general technique to improve the performance and scalability of network simulators by exposing, identifying, and eliminating redundant computation. There are three parts to effectively applying staging to a simulator; namely, *function caching and reuse* to eliminate unnecessary computation, *event-restructuring* to expose redundant computations, and *time-shifting* to enable efficient event processing.

Function caching and reuse forms the foundation of the staging approach. Without loss of generality, each event in a discrete-event simulator can be treated as a sequence of function invocations. A staged simulator caches arguments, results and side-effects of function invocations, and later avoids redundant computations by reusing results and reapplying side effects when the same function is invoked with the same arguments. This space-for-time trade-off ensures that each expensive function invocation is computed at most once. While function caching and reuse is effective at eliminating redundant computations, it is not directly suitable for application in realistic wireless simulators because it is overly sensitive to changes in the arguments to function invocations. In typical wireless simulations, many computations depend on continuously varying inputs, such as the current simulator time, and any change in inputs, no matter how small, precludes reuse of previous results. Similarly, real-valued parameters make it difficult to achieve effective cache hit rates. These properties make naive function caching ineffective for a large class of computations in a simulator.

Event-restructuring improves on function caching by modifying the low-level events in a discrete event simulator such that their results are reusable even when a change in inputs would normally preclude reuse. We introduce three comprehensive techniques, called *currying*, *incremental computation* and *auxiliary results*, for decomposing the events in a discrete-event simulator into a series of events with an equivalent effect. The equivalent transformations retain the semantics of the original event and preserve accuracy. The resulting smaller subcomputations, whose dependencies are better isolated, can then be reused more frequently, leading to significant improvements in simulation speed and scale.

Time-shifting takes advantage of the small, fine-grained events created through event-restructuring by reordering them into an equivalent, but more efficient, schedule. Restructured, fine-grained events have fewer inputs and tend to be time-independent. Consequently, restructuring provides the simulator with freedom to judiciously pick when to schedule these events. Time-shifting can speed up simulations through architectural or algorithmic improvements. First, time-shifting can simply reorder the events into a sequence that is better suited to the machine architecture on which the simulator is running. For instance, time-shifting events to access memory sequentially instead of randomly can speed up execution by taking advantage of memory caching and prefetching. And second, time-shifting and batching similar operations can enable a series of piecemeal, small, consecutive events to be computed by a single, more efficient algorithm. For instance, time-shifting sufficiently many node distance calculations may allow a long sequence of such computations to be replaced with an efficient all-pairs shortest paths algorithm.

Overall, staging relies on function caching and reuse to eliminate redundancies, restructuring to break events into smaller computations that can be reused readily, and time-shifting to pick an efficient schedule for function evaluation. In general, staging can be applied to a single simulation run, which we term *intra-simulation staging*, or across a set of similar simulation runs, which we term *inter-simulation staging*.

While staging is a general technique that can be applied to any discrete-event simulator, in this paper, we focus exclusively on how staging can be used to improve the speed and scale of wireless network simulations. We pick wireless network simulation as our target domain because it poses a worst-case scenario. Since network characteristics can and do change dynamically, simulation state must be recomputed frequently. As nodes move about a simulated field, the network-level topology may change rapidly. Link characteristics, routing information, and network topologies must be maintained and recomputed during the simulation, and mobile nodes must continually update their positions in order to provide accurate information to the packet propagation subsystem. In addition, complex physical models make wireless simulation expensive. Due to the broadcast properties of the wireless medium, a simple packet send operation may involve all nodes in the network. These inherent characteristics of the wireless domain render accurate, fast and scalable wireless network simulation difficult, and make isolating redundant computations a challenge.

We applied staging to **ns2** [The VINT Project. 1995], a well-established simulator whose design is typical of traditional discrete event network simulators. Ns2 is of vital interest to the wireless networking community because it includes support for a wide range of protocols and applications, many of which have been extensively validated against real-world implementations. Our application of staged simulation in the **ns2** simulator confirms the benefits of staging. As a natural consequence of eliminating redundant computation, staging in **ns2** reduced overall run time scaling from $O(N^2)$ in the size of the simulated wireless network to $O(N)$ and improved run time by an order of magnitude over the standard **ns2** implementation. Our applications of staging impact only the internals of the simulation engine, maintain strict compatibility with existing simulation scripts and extensions, and preserve the full accuracy of the original computation. Since staging is complementary to other algorithmic techniques for improving the speed and scale of discrete event simulators, we expect it to yield similar benefits when combined with parallel and distributed simulation techniques.

Overall, this paper makes three contributions. First, it describes a general technique called staging, based on function caching and reuse, for improving discrete event simulator performance and scale without degrading accuracy. It shows how this technique can be systematically applied to a widely used, traditional wireless network simulator, **ns2**, to eliminate certain redundant computations both within a single simulator run and across multiple invocations of the simulator. Second, it provides a formal model for reasoning about the execution of discrete-event simulators, uses this model to derive a precise notion of simulation *equivalence* that is useful for expressing optimizations, and formally defines staging using the model. It also illustrates how previous work on eliminating redundancy in simulations can be unified into this model, and how past work differs from staging. Finally, it shows that staging can significantly improve execution time of network simulators and make feasible the accurate simulation of wireless networks with tens of thousands of nodes.

The rest of this paper is structured as follows. The next section provides a formal model of the operation of a discrete-event simulator and provides background on previous work on reusing past computation in simulators. Section 3 defines staging by showing how function caching and reuse can eliminate redundant computation, how restructuring can expose redundant computation, and how time-shifting can be applied to a restructured simulator. Section 4 shows how intra- and inter-simulation staging can be applied to the core of a wireless network simulator. Section 5 evaluates the performance and scalability benefits of staging in the context of a ubiquitous and mature network simulation engine. Section 6 presents related work, and Section 7 summarizes our contributions.

2. MODEL AND BACKGROUND

2.1 Formal Simulator Model

We formalize a discrete event simulator, without loss of generality, as a stylized state machine. The simulator starts in a state σ_0, and progresses through a sequence of states $\sigma_1 \ldots \sigma_n$. For each state σ_i, the simulator also maintains a set of pending *events*, E_i. In order to compute the next state σ_{i+1}, the simulator chooses from E_i the earliest event, which we denote e_i. The simulator transitions from state σ_i to σ_{i+1} according to the current event e_i, which may modify the state and also schedule or cancel later events. An event e_i may correspond to a simulation-level event, such as a scheduled packet transmission or protocol timer, but it can also be used to represent lower level events of the simulator, such as the sequence of individual function calls or machine-level instructions. We intentionally leave the granularity of these events unspecified, since our techniques apply to any level for which a cache lookup is faster than reprocessing an event, and use the same notation throughout the paper to describe events at multiple levels of abstraction.

Formally, we can express an event e_i as a function

$$\langle \Delta, \Sigma \rangle = e_i(\pi_{e_i}(\sigma_i)).$$

which takes its input from the current state σ_i, and returns an incremental state change Δ and a set of new or canceled events Σ. While the model is described in a functional style, it captures simulations written in procedural languages by mapping hidden dependencies to inputs, and side effects to the incremental state change Δ returned by the function. Note also that the event e_i does not operate on the full simulator state, but rather on a subset of the state selected by a projection function π_{e_i}. After computing Δ and Σ by executing the next pending event e_i, the simulator updates the state according to Δ and the pending event set according to Σ. Formally, we write

$$\langle \sigma_{i+1}, E_{i+1} \rangle = \langle \sigma_i \oplus \Delta, (E_i - e_i) \otimes \Sigma \rangle$$

where the operator $\oplus$ applies the incremental state change Δ to the current state. We view σ_i and Δ each as binding of variable names to values, and define operator $\oplus$ to update existing bindings in σ_i with new values from Δ, and introduce new bindings from Δ for variables that do not yet exist in σ_i. Similarly, Σ contains a set Σ_c of events to be canceled and a set Σ_s of new events to be scheduled. Operator $\otimes$ simply computes $(E_i - \{e_i\} - \Sigma_c) \cup \Sigma_s$, the new set of pending events.

This framework provides us with the foundation to formally characterize simulation optimizations and reason about their correctness. We want to ensure that the results of an optimized simulator are correct, that is, they correspond exactly to the output from the original, unoptimized simulator. Below, we formalize this notion of simulation equivalence, which ensures that the modified simulator faithfully represents the original simulator, and that all useful, operator-visible state will be unchanged.

If π_U selects the set of operator-visible data from a state σ, then we say that a sequence of simulator states $\langle\sigma_0, E_0\rangle, ..., \langle\sigma_n, E_n\rangle$ is equivalent to another sequence $\langle\overline{\sigma}_0, \overline{E}_0\rangle, ..., \langle\overline{\sigma}_m, \overline{E}_m\rangle$ if and only if $\forall i, \pi_U(\overline{\sigma}_{\bar{i}}) = \pi_U(\sigma_i)$. Here, $i = \psi(\bar{i})$ is a mapping from the steps of one simulation to those of the other. This mapping is necessary since an optimized implementation may use more or fewer steps to compute a result as compared to the unoptimized version. Informally, an operator inspecting some subset U of the simulator state will observe the same data and changes to the data over time, regardless of which of two equivalent simulators is being used. Note that, by design, this definition of simulation equivalence does not place any restrictions on the event sets E.

With this formal notation, we are ready to describe previous work, define an initial approach to reuse, and provide a complete description of staging techniques.

2.2 Prefix-based Approaches

Among techniques used to improve discrete event simulator performance or scale are several that attempt to eliminate redundant computation either within or across simulation runs. We discuss three illustrative examples, and show how they can be characterized in our simulator model.

Previous work has suggested a technique called *splitting* [Glasserman et al. 1996], based on common prefixes among multiple runs of the simulator. The technique is based on the observation that if two runs have identical initial state, and the simulator can determine that the first k events of both runs will be identical, then the second simulation can begin in state σ_k saved from the first run. In practice, a first run saves selected checkpoints $\langle\sigma_i, E_i\rangle$ on disk, for selected values of i. A second simulation can be started from any of these saved checkpoints, with new events added or removed from the pending event set E_i. Formally, we can describe this second run as a new simulation, beginning in a configuration

$$\langle\overline{\sigma}_0, \overline{E}_0\rangle = \langle\sigma_k, E_k \otimes \Sigma\rangle$$

where the operator selects the value of k and specifies some change Σ to the pending event set. It is never necessary for the operator to modify the state σ_k, since such a change can always be wrapped in an event and placed at the head of the list of pending events. This splitting technique uses a very restricted notion of state equivalence and therefore cannot be applied beyond the earliest point at which two simulation runs diverge in state.

Cloning [Hybinette and Fujimoto 1997] takes advantage of a more general form of state equivalence. With this technique, two runs of a simulator can share computation for an event e_i so long as the relevant inputs taken from the two simulator states can be assured to be identical. Clearly, if $\pi_{e_i}(\sigma_i) = \pi_{e_i}(\sigma'_i)$, Δ_{e_i} and Σ_{e_i} can be reused from the earlier run. But rather than comparing the inputs $\pi_{e_i}(\sigma_i)$ and $\pi_{e_i}(\sigma'_i)$ directly, cloning relies on partitioning the simulation state into separate components called virtual logical processes.

A logical process is a function ϕ over the state σ_i. In cloning, $\phi(\sigma_i) = \phi(\sigma'_i)$ implies that results can be reused for certain events. Here, two computations share results so long as the virtual logical processes they compute on remain identical. This essentially extends the common prefix technique used in splitting to common prefixes of logical processes, or partial state. Both cloning and splitting typically deal with events only at the level of simulation events, such as packet transmissions and receptions.

The *updateable simulation* [Ferenci et al. 2002] technique reuses state from previous runs of the simulator to implement more efficient, incremental computation in the simulators. The intuition behind updateable simulation is that if the entire state of the first simulation run is known, then subsequent, similar runs might gain performance benefits by incrementally updating the old state to match any new simulation parameters, rather than recomputing the new state from scratch. An initial run of the simulator caches the Δ and Σ tuples from each event taken by the simulator. Subsequent runs can reuse stored results from previous runs if protocol-specific knowledge dictates that the results may be reused in the current run. For instance, specific knowledge of the packet queuing policy may allow subsequent simulator runs to detect when a packet dropped in a previous run will also be dropped in the current simulation run. In such cases, the simulator can re-use the incremental state change Δ computed in the previous run. This approach relies heavily on protocol-specific knowledge and prediction of values that have not yet been computed.

3. STAGED SIMULATION

We introduce staging in our formal model by first generalizing the notion of state equivalence, and using these ideas to show how function caching and reuse can be used in the simulator. We then develop staging by formally defining three restructuring techniques, which are used to expose redundancy in the simulator, and a class of time-shifting optimizations, which take advantage of both restructuring and function caching.

3.1 Function Caching

We can generalize the notion of state equivalence seen in previous work by examining the formal model of a discrete event simulator. According to the model, the following conditions are necessary and sufficient to ensure that two computations will result in identical modifications $\langle\Delta,\Sigma\rangle$ to the current state and pending event set:

(1) $e_i = e_j$

(2) $\pi_{e_i}(\sigma_i) = \pi_{e_j}(\sigma_j)$.

That is, if two events and their inputs are identical, then the results of one event may be used in place of the other. This definition of equivalence can eliminate more computation than prefix-based approaches. At the same time, it avoids the use of protocol-specific knowledge by giving precise conditions for when computation reuse is possible.

The simplest way to implement this approach to eliminating redundant computation is function caching. During a simulation run, the simulator caches the result $\langle\Delta,\Sigma\rangle$ of each event in an *event cache*, a table of results indexed by the inputs to the computation. Before executing an event on new inputs, the event cache is checked to see if the result for the new inputs has already been computed and, if so, the cached result is used in place of the computation. This technique can be extended to span multiple simulation runs by maintaining the event cache on disk.

By itself, this approach has serious limitations in the context of network simulation. First, to avoid excessive overhead, only those results that are likely to be reused and are expensive to compute should be saved. Worse, almost all simulation state is dependent on the clock value t, which typically changes between network events. Consequently, it is rare for two computations to have identical inputs, even between two simulation runs. While function-caching as presented here is strictly more powerful than prefix-based approaches, its limitations must be addressed if it is to be widely applicable.

3.2 Event Restructuring

Staged simulation improves on function caching by enabling optimization even when inputs are not identical across computations. To enable this, we rely on a number of techniques for exposing redundant computation through code restructuring. Using the simulator model, we first look at where computation reuse may be possible and how it can be exposed. In the subsequent sections we give several concrete examples of these techniques, and an evaluation of their application in a popular network simulator.

In order to examine our approach at a high level, let us look closer at an individual simulator event. An individual event e_i takes several arguments as input, say x, y, and z. Function caching is applicable to two computations

$$\begin{aligned} \langle \Delta, \Sigma \rangle &= e_i(\pi_{e_i}(\sigma_i)) = e_i(x,y,z) \\ \langle \Delta', \Sigma' \rangle &= e_j(\pi_{e_j}(\sigma_j)) = e_j(x',y',z') \end{aligned}$$

if and only if $e_i = e_j$, $x = x'$, $y = y'$, and $z = z'$. A parameter mismatch between two invocations, say $z \neq z'$, prohibits the results of previous invocations from being reused.

The first technique used by staged simulation to cope with this problem is based on *currying*, which decomposes the event e_i into two events, say e_i^1 and e_i^2. Conceptually, have smaller functions that can be composed to form the original result $e_i(x,y,z) = e_i^2(e_i^1(x,y),z)$.

$$\begin{aligned} \langle \Delta_1, \Sigma_1 \cup \{e_i^2\} \rangle &= e_i^1(x,y) \\ \langle \Delta, \Sigma \rangle &= e_i^2(c,z) \end{aligned}$$

The first half of the computation, e_i^1, returns an incremental state update with the hidden partial result c, and also schedules the second half of the computation, e_i^2. The second half of the computation then retrieves the saved partial result c, finishes the computation, and returns the final value. Here, we can apply function caching directly to the e_i^1 and e_j^1 computations, since they have identical inputs $x = x'$ and $y = y'$. The second piece of the computation need not be performed immediately following the first, but may be spaced arbitrarily far apart in the sequence of simulator events. This technique can be applied repeatedly to decompose an expensive computation into a tree of cacheable subcomputations.

A second technique is *incremental computation*, which restructures events to reuse the results of previous, nearby computations. For example, wireless simulators typically need to maintain a graph of network connectivity. Each time a node moves, this graph must be refreshed, but it is often much more efficient to incrementally update the previous result than compute a new result. This technique often relies on the continuity of results with respect to some input, which ensures that small changes in the input will lead to only small changes in the result. In our model, incremental computation replaces an event

$$\langle \Delta, \Sigma \rangle = e_i(x, y, z)$$

with an equivalent computation

$$\langle \Delta, \Sigma \rangle = e_i'(x, y, z, \Delta')$$

where Δ' is the result from some previous execution of e_i.

The third and final restructuring technique we introduce is the use of *auxiliary results*. This generalizes the incremental computation approach by reusing not only previous, nearby results, but also *auxiliary results*. In this form, we replace a computation e_i with a new computation

$$\langle \overline{\Delta}, \Sigma \rangle = e_i'(x, y, z, \overline{\Delta}')$$

where $\overline{\Delta}$ contains not only the result Δ needed from the computation, but also auxiliary information or partial results that may be useful to later computations. The later computations can then take this data, and more efficiently compute new results.

One instance of auxiliary results that we have found particularly useful is to compute and save upper and lower bounds, then later refine these bounds into a precise result. For example, in many wireless networks nodes move with some known maximum speed. We can take advantage of this information to reduce the amount of work the simulator has to perform at run time. While computing the set of nodes within range of a sender, we can also mark certain nodes that are sufficiently close to the sender such that they cannot travel beyond his transmission radius even at maximum speed. Later, the same sender can avoid computing distances to the previously marked nodes, since these nodes cannot have moved out of range. This example illustrates that auxiliary results can be used to eliminate redundant computations even when the inputs to the computation are not identical. Formally, this example involves transforming an event

$$\langle \Delta, \Sigma \rangle = e_i(x, y, z)$$

into two events

$$\langle \Delta^1, \Sigma^1 \rangle \cup \langle \Delta^2, \Sigma^2 \rangle = e_i^1(x, y, z) \cup e_i^2(x, y, z)$$

where the results $\langle \Delta^1, \Sigma^1 \rangle$ of e_i^1 are known to be stable so long as the parameters x, y, and z, stay within some predetermined limits. The simulator can then be modified to cache the results of e_i^1, as with the currying method. Later invocations can reuse results so long as the parameters fall within the limits set on each of the parameters.

3.3 Time-shifting

The restructuring techniques described above have the effect of breaking large, monolithic computations with many inputs, into smaller, fine-grained computations with fewer inputs. In addition, the resulting events will tend to have more limited dependencies on simulator state. These properties provide the opportunity to apply time-shifting optimizations. Recall that the mapping function ψ allows the simulation some flexibility in re-ordering function evaluation, so long as the reordering does not affect the operator-visible state of the simulation. We can use this flexibility by computing the results of many events simultaneously, rather than individually and spread out over time. Formally, we reorder events that produce internal results under a mapping ψ, perhaps grouping similar computations together in the sequence of computations. For example, if events a and b have already been decomposed by currying, the simulator might execute the sequence of events

$$...a^1 a^2 ... b^1 b^2 ...$$

during the simulation. Precomputation can transform this to

$$...a^1 b^1 a^2 ... b^2 ...$$

if the inputs to b^1 are unchanged by the reordering.

Reordering events through time-shifting can improve efficiency in two distinct ways. First, we can use time-shifting to gain architectural benefits. By controlling the time at which computations are performed, we indirectly control accesses to disk or memory. This enables staging to spread out load to the disk, reduce working set sizes, or improve memory cache performance. For instance, the memory footprint of a simulator exhibits greater locality when all events for one node are processed in batches, since these computations will typically all operate on the same per-node data structures. Second, we can gain performance through improved algorithms. Specifically, the separate computation of events a^1 and b^1 might combined into a single, more efficient event that computes the results for both computations simultaneously. For instance, if each node computes a shortest-path tree in the network, these individual computations might be shifted to a single point in the simulation. At this point, a single more efficient all-pairs shortest-path computation can replace the individual computations.

4. APPLICATIONS OF STAGING

In this section, we describe how we can apply staging in the context of wireless network simulators to improve their performance and scale. In particular, we have applied staging to **ns2**, a commonly used and well-understood simulator typical of the current state of the art in wireless network simulation. In **ns2** the wireless physical layer and mobility models are the largest consumers of processing time in common simulation scenarios. These computations pose the most significant bottlenecks to efficiency and scale. Consequently, we focus on staging computations related to node mobility and the wireless physical layer throughout this section.

Under typical wireless mobility and physical models, simulators must perform numerous calculations when a packet is transmitted in order to ultimately determine which nodes will receive the packet. These calculations depend strongly on the positions of sending and receiving nodes, packet transmission and reception signal strength thresholds, geography, and radio and antenna models. Consequently, in wireless simulation, neighborhood calculations are fundamental, frequent and expensive.

Mobility computations are an especially challenging domain in which to implement staging, because inputs are dynamically varying and time-dependent. Identical inputs to high-level computations are rarely found within a single run of a simulator, or across multiple similar runs. In addition, the monolithic structure of typical implementations obscures any redundant computations performed at runtime.

We incrementally describe four different applications of staging to **ns2**, each employing a different approach to eliminating redundant computation. The first is an example of decomposing a computation via currying and reusing common intermediate results across events. The second demonstrates the use of upper and lower bounds, or auxiliary results,

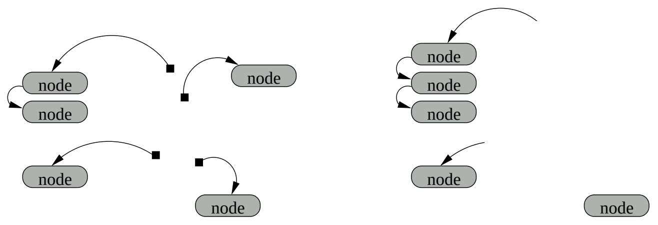

Fig. 1. Grid-based neighborhood computation data structure with either fine or coarse-grained grid granularity. The source node at center is transmitting with radius shown. Examined grid bins are shaded, and linked lists indicated by arrows.

to enlarge the overlap in computation across events. The third optimization illustrates the use of time-shifting as a staging technique, and the final one demonstrates inter-simulation staging by reusing results across multiple similar runs of the simulator. A surprising result is that the final staged simulator looks nothing like the initial simulator implementation, and achieves a dramatic, qualitative improvement in both performance and scale.

4.1 Currying: Grid-based Neighborhood Computation

As an initial, elementary application of staging, we first restructure the neighborhood set computation using a well-known grid-based approach. This computation determines the set of nodes within range of a transmitting node, and hence the nodes that must receive a copy of each packet transmitted by the node. This restructuring is intended to expose the redundancy between and within calls to the neighborhood computation. In our simulator model, we have an event e taking as inputs $\langle t, src \rangle$, where t is the simulator clock value and src is the source node which internally keeps track of its x and y coordinates. Several other parameters, such as the global list of nodes and node positions, and other simulation parameters, are hidden for simplicity. The return value of the event is a pair $\langle dests, \emptyset \rangle$, representing the set of destination nodes within range of the sender, to be passed to the packet transmission routine, and an empty set indicating that this event does not schedule or cancel any other events.

A first example of staging can be seen in a well-known and straightforward grid-based neighborhood computation approach, where we reuse previously computed transmission results for nodes that are close by in distance within a single computation, and expose and reuse partial results between packet transmissions. Grid-based neighborhood computation first divides the coordinate space into a grid of buckets, with each bucket holding a list of nodes positioned within the corresponding grid rectangle. The geographic partitioning and node lists are shown in Figure 4.1. This partial information about node positions can then be used to quickly determine if a group of nodes falls entirely outside the possible transmission range of a node, thereby eliminating the need to perform individual calculations for each node. Nodes in the remaining buckets, which may or may not be in range, are checked individually as before. All of the state needed for the grid is stored in the simulator state variable σ, but outside of the operator-visible portion $\pi_U(\sigma)$.

Grid-based neighborhood computation can be classified as staging in two distinct ways.

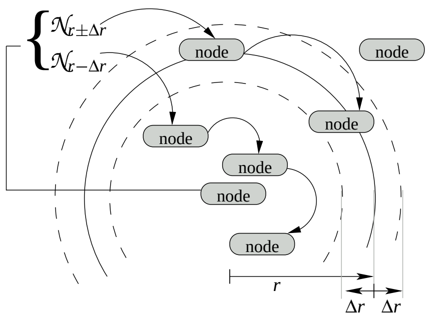

Fig. 2. Neighborhood Cache Entry Structure for Node at Center, Showing Transmission Radius r, $\Delta r = 2s_{max}\Delta t$, and Linked Lists (Arrows) for Sets $\mathcal{N}_{r-\Delta r}$ and $\mathcal{N}_{r\pm\Delta r}$

First, the computation is decomposed into two parts, where the first updates the grid if nodes have moved, and the second uses the updated grid to obtain a precise result. The grid data structure is shared across all computations, and updated whenever needed, either lazily, just before a packet transmission, or continually as nodes move. The second use of staging occurs within a single computation. By grouping nearby nodes into grid buckets, a single distance computation or estimate can be used in place of many individual computations. The benefits of grid-based computation stem from reducing the number of nodes examined on each packet transmission, and from sharing grid maintenance across all packet transmissions.

4.2 Auxiliary Results: Neighborhood Caching

The grid-based approach provides a base to which we can successively apply additional staging. Variations on the grid approach allow more advanced applications of staging using auxiliary results to reduce redundancy in computation across packet transmissions. In common simulation scenarios, inter-packet spacing is very short in comparison to the speed at which nodes move. Depending on node mobility and traffic patterns, many hundreds or thousands of packets may be transmitted from a single node before nodes move a significant distance. That is, we should expect the inputs to, and hence the results of, the grid-based neighborhood computation for a node to be reusable across many packet transmissions.

Since node positions will vary slightly, we should not expect the neighborhood set to be identical to the set computed during the previous packet transmission. However, nodes that were sufficiently far from the sender during the previous transmission will still be out of range of the sender, and nodes that were sufficiently close will still be in range. Computing this additional information at the time of packet transmission is efficient, and this auxiliary data can later serve to quickly eliminate many nodes from consideration in subsequent transmissions. These conservative upper- and lower-bounds on the neighborhood set will remain valid for some time after they are computed, depending on the amount of node mobility and the tightness of the bounds, so we can efficiently check if the saved results are reusable. We therefore restructure the neighborhood set computation to first compute an bounds on the result, with a known expiration time, then refine this bound during each packet transmission into an exact result. After restructuring, a straightforward application of function caching is used to cache and reuse the common auxiliary results, the upper-

and lower-bounds, across many packet transmissions.

This restructuring exposes one additional parameter, Δt, to control the caching policy. This parameter fixes the desired epoch duration for which the bound on the neighborhood set will be valid. If s_{max} is the maximum possible node speed in the movement scenario, then the maximum change in distance between two nodes in an epoch is just $\Delta r = 2 s_{max} \Delta t$. If two nodes are within distance $r - \Delta r$ at some time, then they will remain within range r for Δt seconds into the future. Consequently, the precise position of nodes beyond distance $r + \Delta r$ need not be computed at all for Δt seconds into the future. Thus, neighborhood caching further reduces the number of nodes that need to be examined on every packet transmission, since only nodes in the annulus around the sender could have changed status since the last packet transmission.

We maintain a cache to capture this upper bound on the neighborhood set of each node. At most one cache entry is maintained for each node in the network. A cache entry, illustrated in Figure 4.2, is composed of an expiration time and two sets, $\mathcal{N}_{r-\Delta r}$ and $\mathcal{N}_{r\pm\Delta r}$, containing lists of the nodes within a ball of radius $r - \Delta r$ and those in the annulus with radii $r \pm \Delta r$. During packet transmission, the cache manager computes the set of nodes within range of a given node by first looking for a valid cache entry. Finding an entry that has not yet expired, it can immediately consider all nodes in the list $\mathcal{N}_{r-\Delta r}$ to be within range. The second list $\mathcal{N}_{r\pm\Delta r}$ is then scanned, and each node found to be within range is appended to the final result. At the same time, it can cheaply but conservatively update the lists, moving some nodes from $\mathcal{N}_{r\pm\Delta r}$ to $\mathcal{N}_{r-\Delta r}$ and eliminating others from $\mathcal{N}_{r\pm\Delta r}$ entirely. If, on the other hand, no cache entry is found during packet transmission, the cache manager consults the underlying grid and constructs a cache entry with expiration Δt seconds into the future.

In the above caching scheme, there is some additional overhead during cache misses, when computing $\mathcal{N}_{r\pm\Delta r}$, since a larger radius is considered than previously necessary. This overhead is controlled directly with the parameter Δt, which fixes the longevity and the accuracy of cache entries. In addition, there is overhead associated with scanning the list of nodes in $\mathcal{N}_{r\pm\Delta r}$ during each cache hit, but this is also limited by appropriately choosing the Δt parameter. We characterize these overheads experimentally in Section 5.

In terms of the simulator model, this type of staging eliminates redundant computation in only one particular case. If the inputs to two neighborhood computations are $\langle t, src \rangle$ and $\langle t', src' \rangle$, neighborhood caching addresses the case when $t' \approx t$ and $src' = src$. That is, a cached result from a packet transmission is reused only when transmitting from the same source at a nearby time. The caching presented here is only applied when both computations occur within the same simulation run.

4.3 Time-shifting: Perfect Caching

We can expand the overlap in computation by looking for other redundancies than that addressed by neighborhood caching. If two neighborhood computations have inputs $\langle t, src \rangle$ and $\langle t', src' \rangle$, as before, we now explore the case when $t = t'$ and $src \neq src'$. That is, we look for redundancy when two computations have the same clock parameter t but differing source parameters.

It is unlikely that two nodes will transmit packets at identical times. However, neighborhood cache entries can be constructed at any time and take as input the same parameters $\langle t, src \rangle$. There is a large overlap in computation when constructing multiple cache entries independently. In the worst case, each pair of nodes will be considered twice, as the receiver of a packet does not take advantage of the distance computed by the sender. This is especially acute in light of the rapid sequences of packet exchanges between a pair of nodes, which are common in wireless MAC and network protocols. We use time-shifting to address this redundancy by reordering the creation of cache entries.

A staged simulation approach, which we term *perfect caching*, eliminates this redundancy by precomputing all cache entries simultaneously, thereby forcing the condition that $t = t'$. This approach maintains the same data-structures as neighborhood caching. But, rather than calculating cache entries on-demand, it precomputes all cache entries at the beginning of every Δt epoch. All normal queries for neighborhood information are then guaranteed to be satisfied from the cache. There are two potential advantages to batching computation in this way. First, by grouping all accesses to the mobility data-structures, the simulator might achieve higher memory locality and improved working set sizes. And second, we can implement a more efficient algorithm that simultaneously computes all of the cache entries together, rather than individual computations for each entry. Specifically, the positions of all nodes can be updated only once per epoch, just before the single computation, and each pair of nodes need be examined at most once per epoch, rather than twice.

The overhead of this technique is a scheduled event during each Δt epoch. Also, perfect caching may introduce additional, wasted computation if some nodes do not send packets during an epoch, and thus do not use their cache entries. In a sparse or quiet network, perfect caching might construct more entries than needed during the simulation. This potential for wasted work can be addressed by appropriately choosing the Δt epoch parameter.

The underlying grid is normally maintained during each epoch as nodes move about the geography. At each time a node moves across a grid boundary, an event must be scheduled to update the affected grid cells. In many networks this maintenance can be very expensive. However, the perfect caching approach accesses the underlying grid only once per epoch, opening the possibility that the grid might be re-created from scratch once per epoch, rather than maintained and updated from one epoch to the next. Staging introduces a new trade-off in choice of maintenance versus re-creation. This trade-off depends on the size of the network, which impacts the cost of grid re-creation, and the amount of mobility, which impacts the cost of maintenance. We explore both these alternatives for typical network scenarios in Section 5.

4.4 Inter-simulation Reuse: On-disk Caching

A final inter-simulation staging application improves on perfect caching and demonstrates how staging can be applied across multiple similar runs of the simulator. Inter-simulation staging expands the scope of caching and reuse to span multiple runs of the simulator by using the disk as a persistent cache. In the first run of the simulator, cache entries are generated and expire exactly as in intra-simulation staging. As cache entries expire, however, they are written to disk for use in later simulator runs. Disk access can be done efficiently in the background by introducing a moving window and a worker thread. The moving window covers cache entries that have expired, but have not yet been written to disk. The worker thread, operating entirely in the background, can spool these entries to disk, then expunge them from the cache. Subsequent runs of the simulator perform a complimentary procedure. Here, the worker thread prefetches cache entries in the background, saving them into a sliding window just ahead of the simulation clock. The simulator can then use these cache entries directly until they expire and are immediately expunged. Again, all disk accesses are performed in the background.

We apply inter-simulation staging to the perfect caching scheme described above. This takes advantage of common simulator usage, where a batch of simulator runs often share a single mobility scenario. In this case, perfect caching will perform identical work during each simulation run, since cache entries are computed at pre-chosen times independent of the network load or other simulator parameters. The first run of the simulator saves all neighborhood cache entries to disk, and the second and subsequent runs avoid recomputing these entries by reading the results directly from disk.

The amount of computation saved in this manner is potentially significant. The intra-simulation examples above, while reducing the amount of computation during packet transmission, also add some additional work to maintain or re-create the grid for use during cache misses. In particular, many grid-crossing events might be scheduled in the event queue, leading to more work in the event scheduler and dispatcher. This application of inter-simulation staging builds on the intra-simulation staging techniques by reducing the number of scheduled events generated by the grid manager and the cost of constructing neighborhood cache entries in the perfect caching scheme. Surprisingly, in the second and subsequent runs of the simulator we can eliminate the underlying grid entirely, as well as all the work for constructing cache entries, since all requests will be satisfied by cache entries read from disk.

Once an on-disk cache has been constructed, subsequent runs of the simulator do not maintain a grid, do not need to track changes to node positions, and require no scheduler events. Eliminating this overhead from simulations can lead to significant speedups.

5. EVALUATION

We have implemented each of the optimizations described in the previous section in the **ns2** simulator. Overall, we find that even the simplest application of staging reduces the run time of the simulator significantly, and allows for practical simulation of much larger network sizes than previously feasible. We show that the additional applications of intra-simulation staging improve the robustness of the results. In particular, the initial, grid-based staging is very sensitive to the choice of granularity, while the additional applications of staging eliminate this sensitivity and work well even with a poor choice of parameters. The application of inter-simulation staging improves speed and scale yet further. With the final staged implementation, we regularly simulate networks of over 1000 nodes in the time it previously took to simulate networks of only 200 nodes.

In addition to evaluating the performance and scalability benefits of staging, we also characterize the effect of each parameter we have introduced. We show that it is possible to easily or automatically find near-optimal choices for these parameters and, at the very least, avoid parameter choices that would lead to run time behavior worse than the default, non-staged implementation.

5.1 Evaluation Platform and Environment

We take as our baseline the **ns2** version 2.1b9a simulator. Most simulations were completed on a single-processor machine equipped with 1.7GHz Pentium 4 processor and 256MB of physical memory. Physical memory is an important constraint in **ns2**; more generous machines can simulate proportionally larger networks before becoming memory-limited. We discuss the memory requirements of staging in detail in Section 5.5.

Before implementing our staging techniques, we made a few non-standard modifications

Table I. Default Simulation Parameters for Experiments

Network load	
Model	CMU constant-bitrate
concurrent data streams	30
Packet size & rate	512 bytes $\times$ 8 packets/s
Node mobility	
Model	CMU random-waypoint
Maximum node speed	5 m/s
Pause time	10 s
Field density	$\approx$ 30 nodes / km^2
Simulation	
Routing protocol	AODV
MAC layer	802.11 (CMU)
Transmission and interference radius	551 meters
Simulation time	400 s

to improve the baseline **ns2** code. First, we disabled all unused packet headers to improve memory locality. Second, we implemented more memory-efficient packet tracing. Third, we implemented a more sophisticated calendar-queue event scheduler in order to reduce the sensitivity of the simulator to minor changes in the event distribution and to better optimize several common-case scenarios. And finally, we enabled standard compile-time optimizations throughout the simulator and runtime libraries. These changes improved the run time of the baseline simulator by approximately 85% and substantially reduced its memory requirements. Thus our results for staging are compared to an aggressively optimized baseline implementation representative of the current state of the art in wireless network simulation.

Our simulation setup closely resembles that of [Broch et al. 1998], an exemplar of common scenarios used in wireless networking simulation. While we use the standard CMU Monarch mobility extensions, the communication model generators from the standard **ns2** distribution, and the AODV ad hoc routing protocol implementation, our results are not specific to these choices of application, mobility model, or communication pattern. These system parameters, summarized in Table I, closely follow the standard values used in ad hoc networking literature. Note that we use a node radius of 551 meters, rather than the nominal packet reception range of 250 meters, to properly account for any possible interference effects.

We evaluate simulator performance with several levels of staging, corresponding to each successive application of staging described earlier. The different levels are detailed in Table II. Note that we evaluate perfect caching using both the grid maintenance (L_{3a}) and grid re-creation (L_{3b}) strategies, and that inter-simulation L_{4a} staging is based on the L_{3a} grid maintenance strategy. We generate five random network scenarios for various network sizes, and evaluate each simulator implementation on all five networks. Except where noted, each data point represents the average execution time of the simulator in all five scenarios. The standard deviation in relative execution time of the simulators on any given network is small, less than 0.2% in all cases.

5.2 Simulator Performance

We first examine how the different applications of staging affect total simulation execution time using a 1500 node network. Due to the large network size and excessive execution time required by the baseline simulator, this experiment is conducted on a high-

Table II. Levels of Ns2 Optimization for Experiments

Level	Optimizations
L_0	Ns2 baseline
Intra-simulation staging	
L_1	L_0 + Grid-based
L_2	L_1 + Neighborhood Caching
L_{3a}	L_2 + Perfect caching (grid maintenance)
L_{3b}	L_2 + Perfect caching (grid re-creation)
Inter-simulation staging	
L_{4a}	L_{3a} + On-disk caching (generation)
L_{4b}	L_0 + On-disk caching (use)

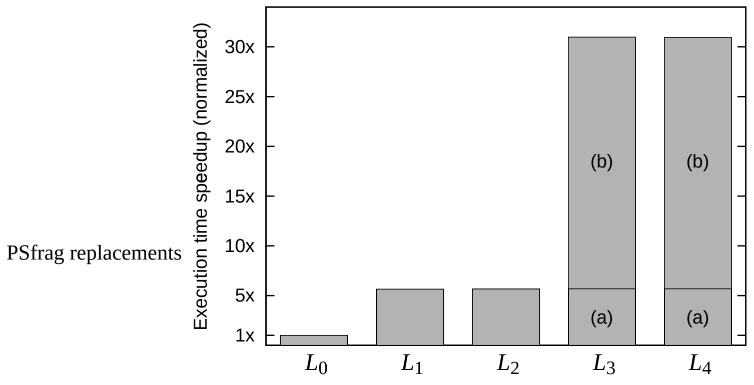

Fig. 3. Speedup in Execution Time with Increasing Staging Relative to Baseline Ns2 Implementation using a 1500 Node Network

performance server-class machine, and only performed for a single network scenario. In this experiment, we fix grid granularity at 250 meters and Δt at 2 seconds, and later describe their selection and the sensitivity of staging to these parameters. The speedup achieved by increasing levels of staging relative to the baseline simulator is shown in Figure 3.

These results demonstrate the dramatic improvement in simulator speed that can be achieved with staging. The final staged simulator, shown as L_{4b}, improves run time by a factor of more than 30 over the optimized baseline by caching results across simulations. Staging achieves more than a factor of 50 speedup over the stock, unoptimized **ns2** implementation. This improvement stems from almost entirely eliminating the work required for neighborhood set computations through staging. The performance of the inter-simulation staging approach shows that the disk accesses required by this technique can be pipelined and executed in the background, and so do not visibly impact the performance.

Depending on the network scenario, and the trade-off in grid recomputation versus grid maintenance, the L_{3b} simulation technique can also do just as well as inter-simulation staging. In this scenario, grid maintenance is expensive, but the cost of grid reconstruction is very cheap, comparable to the cost of reading previously computed results from disk.

PSfrag replacements

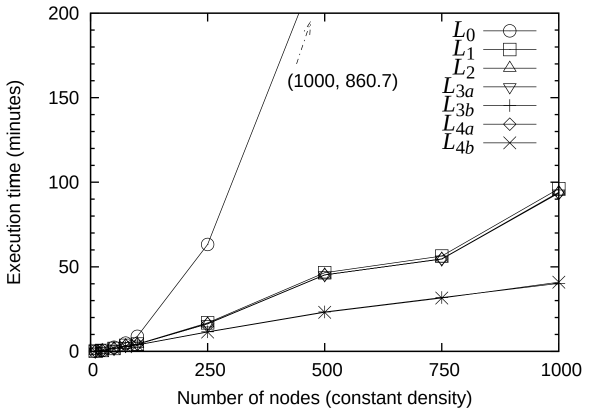

Fig. 4. Effect of Network Size on Total Simulation Run Time Holding Node Density Constant

5.3 Scaling with Network Size

In order to evaluate how staging affects simulation scale, we simulated networks with varying number of nodes while holding the application-level load constant and increasing the field size to maintain a constant node density.

Figure 4 shows that staging can dramatically and qualitatively improve the scalability of wireless simulators by reducing redundant computations. The graph shows that staging changes the scaling behavior of the simulator from quadratic to linear in the size of the network. This experiment also demonstrates the benefits of inter-simulation L_{4b} staging and intra-simulation L_{3b} staging, which achieves more than a 2-fold improvement over other techniques by eliminating grid maintenance. In the inter-simulation case, access to the event cache stored on disk replace grid computations, while the intra-simulation case reconstructs the grid whenever needed. Both of these approaches have negligible cost. Using these techniques, we have performed simulation runs as large as ten thousand nodes in about 10 hours under the load conditions described earlier.

Although several intra-simulation staging approaches show similar performance in this experiment, each of the restructuring steps were necessary to achieve the final benefits of inter-simulation staging. In addition, the various intra-simulation techniques exhibit different behaviors as optimization parameters or network characteristics change. As we show in the next two sections, the more advanced optimizations offer increased robustness and stability, an advantage not evident in Figure 4.

5.4 Optimization Parameters

It is important to characterize the effect of any new simulation parameters introduced by our optimization techniques. We study simulation performance under various choices for optimization parameters and examine the robustness and stability of the different optimization levels. Recall that the grid-based intra-simulation approach uses a specified granularity g, and the caching intra-simulation approach uses epoch length Δt.

5.4.1 *Grid Granularity.* We first evaluate the effect of varying grid granularity on each level of staging. Intuitively, it is clear that a very fine granularity will give rise to many

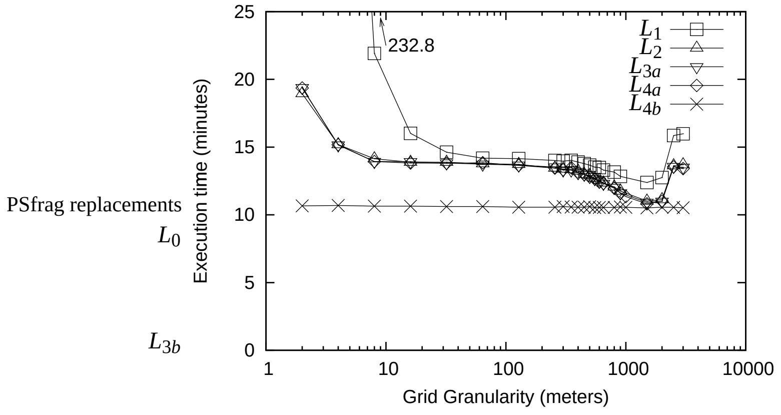

Fig. 5. Effect of Varying Grid Granularity on Simulation Run Time

grid-crossing events as nodes move about in the topology, and will also lead to more work in packet transmission since many empty bins will be scanned for nodes. Conversely, a very coarse granularity reduces to a single bin and, essentially, a scan over all nodes during each packet transmission or cache miss. The optimal choice resides in between these two extremes and depends on the particular choice of simulation parameters.

We examine the impact of grid granularity on simulation time experimentally. We run the simulator with the same configuration as before, using a 250 node network and with Δt fixed at 2 seconds, but vary the grid granularity. Figure 5 shows the effects of grid granularity on simulator performance.

This experiment shows that nearly all of the degradation due to a poor choice in granularity is mitigated by the use of the higher levels of staging. In these cases, the grid is consulted only in the rare case of a cache miss. Consequently, grid performance is not critical to overall performance, and staging masks the impact of a poorly tuned grid. This enables the simulator to be more flexible in the choice of granularity. For example, memory considerations might dictate that a coarse-grained grid be used even when a fine-grained grid would be more efficient. The higher levels of staging, which are less sensitive to the granularity, are able to make these kinds of trade-offs without substantially impacting performance. The grid-based L_1 scheme, on the other hand, must be very careful not to choose sub-optimal parameters.

It is interesting to note that even the right-most extreme of L_1 staging performs much better than the **ns2** baseline implementation, even though they both perform a complete pass over all nodes during each packet transmission. The difference in performance is due to a difference in packet delivery. The L_1 implementation creates copies of each packet only for the nodes in range of the sender, while the baseline simulator creates a copy for every node in the network.

5.4.2 *Epoch Length.* The accuracy and overhead of constructing cache entries is controlled by the Δt parameter for neighborhood caching. Recall that Δt specifies the desired expiration time when constructing a cache entry. A larger value means that a larger radius must be examined to build a cache entry, which leads to larger and more imprecise cache

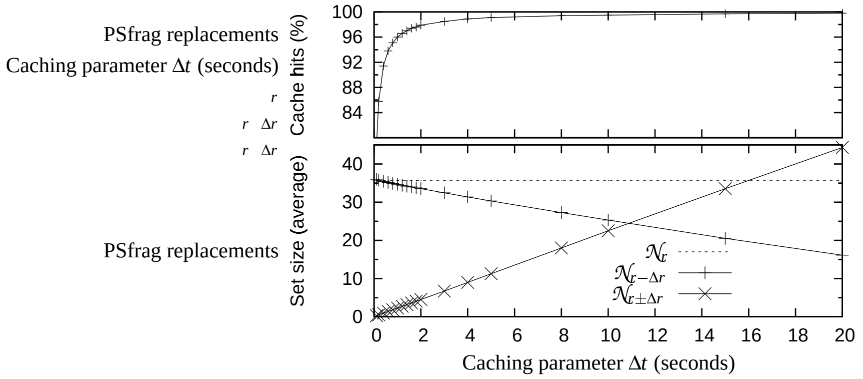

Fig. 6. Effect of Varying Caching Parameter Δt on Cache Hit Rate and Neighborhood Sizes

values, but allows the entry to remain valid for longer.

We set up our simulator as the previous experiment, but fix the grid granularity at 250 m. Figure 6 shows how Δt controls the cache hit rate (top), and the sizes of the two neighborhoods sets $\mathcal{N}_{r-\Delta r}$ and $\mathcal{N}_{r\pm\Delta r}$ stored in cache entries (bottom). We only show the results for L_2 caching; those for L_{3a} perfect caching and the first phase L_{4a} of intra-simulation staging are nearly identical. For reference, the actual average neighbor set size for queries is shown as constant $\mathcal{N}_r$.

The overheads associated with caching are limited by the cache hit rate and $\mathcal{N}_{r\pm\Delta r}$. A very small value for Δt leads to many cache misses, each of which is potentially expensive. Conversely, a large value for Δt forces both cache hits and misses to process a larger set $\mathcal{N}_{r\pm\Delta r}$. The cache is effective for reasonable values of Δt, roughly 2 to 4 seconds, with high hit rate but still reasonably sized $\mathcal{N}_{r\pm\Delta r}$. The curves for the neighborhood set sizes can be explained geometrically based on the known transmission radius, and the average number of neighbors of transmitting nodes. The cache hit rate is a function of the average inter-packet spacing. While our implementation does not pick Δt automatically, the figure shows that a near-optimal value for parameter Δt can be computed as a function of the packet rate, node density, and transmission radius.

Surprisingly, even with such varying cache behavior there is very little overall change in total simulation run time. The data underlying Figure 6 shows that over the entire range of values for Δt run time varies by at most 5%. The L_2, L_{3a}, and L_{4a} staging levels all perform similarly, while the second phase L_{4b} inter-simulation approach improves run time by approximately 30% as compared to L_{3a}, independent of the Δt parameter. As with the grid granularity, nearly any reasonable choice of parameter value for Δt will work well for the highest levels of staging.

5.5 Memory Utilization and Performance

Staging is based on a fundamental trade-off between caching and reuse in memory versus recomputation at the CPU. On the one hand, memory is often a scarce resource in discrete event simulators, so it is important that memory use from staging is controlled and limited. The **ns2** simulator in particular is very memory intensive, and available memory severely constraints the size of the simulated network.

On the other hand, staging can reduce memory requirements through time-shifting; that is, by scheduling memory intensive events efficiently. In this section we detail the memory requirements of staging as compared to the original, non-staged simulator, and describe the cache management policies for the different applications of staging implemented in **ns2**. To provide perspective, we note that even after applying traditional optimization techniques to reduce memory consumption, more than 30 KB is required for each simulated node in the baseline simulator.

Overall, none of the staging applications has more than a modest impact on memory consumption. In the experiments described above, a grid using a granularity of 250 meters requires between 1 and 4 KB total memory, depending on the size of the geographic field. For a grid with 10 meter granularity, this can rise to as much as 1.2 MB for the larger networks. As an extreme case, a grid of 1 meter granularity would require well over 100 MB.

Neighborhood caching L_2 staging, perfect caching L_{3a} and L_{3b} staging, and the first phase L_{4a} inter-simulation staging require additional memory, beyond that required by the grid-based approach, for storing per-node cache entries. The size of each cache entry depends on the number of nodes near each potential sender, but, as shown earlier in Figure 6, the cache entry size is modest for a wide range of parameters. Since cache entries are generated during each epoch, the cache is potentially unbounded. However, only the most recent entry is likely to be reused in the near future, so we can limit the cache to only a single entry per node in the intra-simulation staging approaches. In order to save older entries for use across simulations, we employ a sliding-window approach to write recent entries to disk in the background before expunging them from the event cache. In all, across the range of values explored in the above experiments, between 20 and 200 KB total memory is needed on top of the memory needed for the L_1 grid, and this scales linearly with the size of the network.

The second phase L_{4b} inter-simulation staging reduces memory consumption relative to the intra-simulation techniques, since the L_1 grid is no longer needed during this phase. In all but the first run of a batch, only 20 to 200 KB total memory above the baseline **ns2** was needed for the above inter-simulation experiments.

In summary, the impact of staging on memory consumption can be made negligible in all cases we have examined by appropriately choosing staging parameters and implementing effective cache management strategies.

6. RELATED WORK

Several domain-specific examples of staging can be found in existing simulators. In our analysis of the **ns2** implementation, we identified some limited applications of staging, but the technique is neither systematically applied in the implementation nor recognized in the literature. There has been no prior recognition or development of the technique of staging as a general approach to simulation optimization.

Grid-based neighborhood computations are a well-known technique, and represent a limited application of staging. The default **ns2** implementation also contains a grid-based scheme for computing neighbor-sets. A key difference between the **ns2** implementation and our L_1 scheme is that we expose and explore the parameter space of grid granularities, while the previous attempt uses a hard-coded granularity of 1 meter. In typical scenarios, this choice leads to performance worse than the baseline, and is consequently disabled by default. Similarly, Wu and Bonnet [2002] propose an alternative packet transmission routine for **ns2**, essentially equivalent to our L_1 staging with granularity parameter ∞. Our evaluation indicates that this choice of granularity is also particularly inefficient compared to nearly any other choice. These examples illustrate the importance of properly characterizing staging parameters and relating them to system variables such as the transmission radius and expected number of neighbors.

In the context of discrete event simulators, we find occasional use of staging or similar techniques to improve performance. Splitting [Glasserman et al. 1996], cloning [Hybinette and Fujimoto 1997] and updateable simulations [Ferenci et al. 2002] are three related techniques, previously described in detail in Section 2, which eliminate identical computations in multiple runs of the simulator. These techniques do not exploit redundant computations within a single run of the simulator, nor do they address computations that are similar but not identical.

Boukerche et al. [1999] propose a two-phase design for Personal Communications System (PCS) network simulation using SWiMNet. This design is complementary to our use of staging, since it is used to facilitate various lookahead optimizations in a parallel simulation engine, rather than to eliminate redundant computation or optimize multiple runs of the simulator.

The NixVector [Riley et al. 2000] approach improves routing efficiency in the **ns2** simulator by computing and caching routes on demand rather than maintaining a complete routing table. This approach has not been applied between multiple runs of the simulator, nor does it eliminate redundant computations when inputs vary slightly between simulations.

Neighborhood computation in wireless networks resembles the colliding pucks problem, which is well-studied in the literature [Hontalas et al. 1989; Lubachevsky 1990]. Many different approaches have been proposed, some of which use sectoring or grid-based approaches, but none investigate the elimination of redundant computation through caching and reuse.

Various research efforts are underway to improve the scale and performance of discrete-event simulators through distributed simulation. These efforts improve simulation performance and speed by parallelizing the simulation task and carrying it out in parallel on a cluster (for example [Fujimoto 1990; Boukerche et al. 1999; Liu and Nicol 2001; Liu et al. 2001; Liljenstam et al. 2001]), sometimes leveraging specialized language features to reason about the behavior of simulated nodes (for example [Zeng et al. 1998]). Overall, these approaches to high performance simulation are complementary to ours, since staged simulation can be applied equally well in distributed simulators to eliminate redundant operations within a single physical node.

Interestingly, certain kinds of redundant computations have been used in optimistic simulators (e.g. [Jefferson 1985]) to improve simulation scalability. Specifically, optimistic simulation enables concurrent nodes to advance simulation time independently, and may abandon and recompute parts of the simulation state upon receiving a packet whose timestamp is behind the virtual time of a given node. Staged simulation does not address this kind of redundancy, as eliminating this type of redundant computation would cancel the benefits of optimistic simulation.

Some techniques, such as model abstraction and approximation [Huang et al. 1998; Gadde et al. 2001], have been used to reduce simulation run time by trading off accuracy for speed. Our approach differs from model abstraction in that staged simulation results are equivalent to an execution of the unoptimized simulator, and we do not approximate or alter the final result of computations in any way.

Finally, we note that staging has been employed in other settings, most notably in programming languages [Acar et al. 2003] and iterative programming [Liu et al. 1996], to achieve performance. We share with these approaches the fundamental insight that caching and reusing previously computed results can improve performance by eliminating redundant computations.

7. CONCLUSIONS

In this paper, we introduced a formal model of operation for discrete-event simulators, both to characterize previous work on simulator optimizations and to derive a precise notion of simulation equivalency. We then used the model to develop a general technique, staged simulation, for improving the speed and scale of discrete event simulators.

The central idea behind staging is to eliminate redundant or partially redundant computations typically encountered in simulations. Staging relies on caching and reusing partial results to eliminate redundancy. We introduce three different techniques, called currying, incremental computation and auxiliary results, to enable caching and reuse by exposing and isolating redundant computations within a single run as well as across multiple runs of a discrete-event simulator. In addition to eliminating redundant computation, additional performance can be gained in a staged simulator through the use of time-shifting optimizations to alter the time at which computations are performed. Staging is a general technique, retains the original accuracy of an unoptimized simulator and is applicable to a wide range of simulators, including parallel and distributed simulation engines.

Finally, our implementation of staging in the widely used **ns2** simulator shows that staging is an effective technique for reducing simulation run time and improving scalability. We implement three types of intra-simulation staging in **ns2** and enable function reuse between multiple simulations. Overall, our modifications to the standard **ns2** simulation improve the total runtime of the simulator from $O(N^2)$ to $O(N)$. For a specific datapoint involving 1500 nodes, our final staged simulator is over a factor of 50 faster than the stock, unoptimized **ns2**. Intra-simulation staging accounts for a factor of 5 to 30 speedup, while inter-simulation staging achieves a factor of 30 speedup across a range of network parameters. Staging correspondingly improves simulator scalability from quadratic to linear in the size of the network and enables the simulation of networks of tens of thousands of nodes. We find that these techniques are robust in the choice of parameters, and the parameters appear easy to estimate automatically as a function of other simulation variables and observed runtime behavior.

REFERENCES

ACAR, U. A., BLELLOCH, G. E., AND HARPER, R. 2003. Selective memoization. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*.

BOUKERCHE, A., DAS, S., FABBRI, A., AND YILDIZ, O. 1999. Exploiting model independence for parallel PCS network simulation. In *Proceedings of the Workshop on Parallel and Distributed Simulation (PADS)*.

BROCH, J., MALTZ, D. A., JOHNSON, D. B., HU, Y.-C., AND JETCHEVA, J. 1998. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the Conference on Mobile Computing and Networking (MobiCom)*. 85–97.

FERENCI, S., FUJIMOTO, R., AMMAR, M., PERUMULLA, K., AND RILEY, G. 2002. Updateable network simulations. In *Proceedings of the Workshop on Parallel and Distributed Simulation (PADS)*.

FUJIMOTO, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM 33,* 10 (Oct.), 30–53.

GADDE, S., CHASE, J., AND VAHDAT, A. 2001. Coarse-grained network simulation for wide-area distributed systems. In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference*.

GLASSERMAN, P., HEIDELBERGER, P., AND SHAHABUDDIN, P. 1996. Splitting for rare event simulation: Analysis of simple cases. In *Proceedings of the Winter Simulation Conference*.

HONTALAS, P., BECKMAN, B., DILORETO, M., BLUE, L., REIHER, P., STURDEVANT, K., WARREN, L. V., WEDEL, J., WIELAND, F., AND JEFFERSON, D. 1989. Performance of the colliding pucks simulation on the time warp operating system, part 2: Asynchronous behavior and sectoring. In *SCS Multiconference on Distributed Simulation*.

HUANG, P., ESTRIN, D., AND HEIDEMANN, J. 1998. Enabling large-scale simulations: Selective abstraction approach to the study of multicast protocols. In *Proceedings of the Intl. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 241–248.

HYBINETTE, M. AND FUJIMOTO, R. 1997. Cloning: a novel method for interactive parallel simulation. In *Proceedings of the Winter Simulation Conference*.

JEFFERSON, D. R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems 7,* 3, 404–425.

LILJENSTAM, M., RÖNNGREN, R., AND AYANI, R. 2001. MobSim++: Parallel simulation of personal communication networks. *IEEE DS Online 2,* 2.

LIU, J. AND NICOL, D. 2001. DaSSF 3.1 user's manual. Available at: `http://www.cs.dartmouth.edu/research/DaSSF/papers/dassf-manual-3.1.ps`.

LIU, J., PERRONE, L., NICOL, D., LILJENSTAM, M., ELLIOTT, C., AND PEARSON, D. 2001. Simulation modeling of large-scale ad-hoc sensor networks. In *Proceedings of the European Simulation Interoperability Workshop*.

LIU, Y., STOLLER, S., AND TEITELBAUM, T. 1996. Discovering auxiliary information for incremental computation. In *Proceedings of ACM SIGPLAN*.

LUBACHEVSKY, B. 1990. Simulating colliding rigid disks in parallel using bounded lag without time warp. In *SCS Multiconference on Distributed Simulation*.

RILEY, G. F., AMMAR, M. H., AND FUJIMOTO, R. 2000. Stateless routing in network simulations. In *Proceedings of the Intl. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 524–531.

THE VINT PROJECT. 1995. Ns-2 network simulator. Available at: `http://www.isi.edu/nsnam/ns`.

WALSH, K. AND SIRER, E. G. 2003. Staged simulation for improving the scale and performance of wireless network simulations. In *Proceedings of the Winter Simulation Conference*.

WU, S. AND BONNET, C. 2002. An alternative packet transmission procedure for mobile network simulation. In *Proceedings of the Intl. Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*.

ZENG, X., BAGRODIA, R., AND GERLA, M. 1998. GloMoSim: A library for parallel simulation of large-scale wireless networks. In *Proceedings of the Workshop on Parallel and Distributed Simulation (PADS)*. 154–161.

Received May 2003; revised December 2003; accepted April 2004