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 the amount of redundant computation within a simulation. 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 it can improve execution time by a factor of 40 and make feasible 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; D.1.m [**Programming Techniques**]: Miscellaneous

General Terms: Performance

Additional Key Words and Phrases: Staged simulation

1. INTRODUCTION

Network simulations are critical for the design and evaluation of distributed systems and networking protocols. It is crucial to evaluate new systems and protocols across a range of deployment scenarios, especially in newly emerging domains such as mobile ad hoc and sensor networks. Such 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 simulator 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 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, frequently-used primitive in wireless simulations that is invoked each time a packet is broadcast. It computes the relative positions of the nodes in the near vicinity, calculates the power level of the signal at each receiver, and determines the set of nodes at which the signal is above a threshold value. A traditional wireless network simulator that computes the neighbors of a wireless node at time t may *re*compute the neighbor-set at time step $t + \epsilon$ because the nodes may have moved, and hence the neighbor-set changed, during the interval ϵ. While node positions and consequently packet reception strengths vary with each increment of the simulator timeline, the final neighbor-set changes relatively infrequently over small time scales. Much of the time spent re-computing packet reception strengths and re-deriving neighbor-sets is effectively wasted.

A second class of redundant computation occurs between multiple runs of a simulator, and stems from lack of retained information from past runs. Extensive simulation studies, such as protocol comparisons or system evaluations, require hundreds of simulation runs, each with slightly varying parameters. While each run may differ from the others in inputs such as the network traffic pattern or the protocol implementation details, there is often 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 functions from scratch.

This paper describes *staged simulation*, a general technique to improve the performance and scalability of network simulators by exposing, identifying, and eliminating sources of redundant computation. To facilitate this, we first expose redundant computations by partitioning the simulation engine into a collection of *staged computations*, then eliminate redundancy by caching and reusing similar results across stages. We seek to identify when final results, intermediate results, or auxiliary results of a previous computation can be reused later for performance improvements. 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*.

Staging involves restructuring the events in a discrete-event simulator into an equivalent set of sub-computations, caching their results, and reusing them whenever matches are identified. Without loss of generality, we treat each event in a discrete-event simulator as a sequence of function invocations. A first-cut approach for eliminating redundancy is *function caching*. That is, the simulator can cache arguments, results and side-effects of function invocations, and later avoid expensive function invocations by reusing results and reapplying side-effects when the same function is invoked with the same arguments.

While function caching and reuse is effective at eliminating certain types of redundant computations and forms the foundation of our approach, it is not suitable for application to realistic wireless simulators because it is very sensitive to changes in the arguments to function invocations. In typical wireless simulations, most events depend on the simulator time and node positions, and slight differences in any of these arguments render the cached values unusable.

Staging significantly improves on function caching by introducing three techniques, called *currying*, *refinement* and *precomputation*, for restructuring events such that their results are reusable even when a change in input would normally preclude reuse. All three of these techniques operate by decomposing the events in a discrete-event simulator into a series of events with an *equivalent* effect. These smaller subcomputations, whose dependencies are better isolated, can then be reused more frequently, leading to significant improvements in simulation speed and scale.

While staging is a general technique that can be applied to any discrete-event simulator, we focus exclusively on wireless network simulations in this paper, and apply staging to a wireless network simulator. We picked wireless network simulations because they pose a worst-case scenario for staging. Because mobile ad hoc networks have have highly dynamic characteristics, simulation state must be recomputed dynamically and often. 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. For all of these reasons, accurate, fast and scalable simulations of wireless networks is difficult, and isolating redundant computations is a challenge.

We rewrote the event processing engine of ns2 [The VINT Project. 1995], a well-established simulator whose design is typical of traditional discrete event network simulators, to perform staging. 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 quadratic in the size of the simulated wireless network to linear, and improved run time by an order of magnitude over the standard ns2 implementation. Staging impacts only the internals of the simulations engine, maintains strict compatibility with existing simulation scripts and extensions, and preserves the full accuracy of the original computation. Staging is complementary to other techniques for improving the speed and scale of discrete event simulators. Specifically, we expect staging to yield similar benefits when combined with parallel and distributed simulation techniques.

This paper makes three contributions. First, it describes a general technique 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. It also illustrates how previous work 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 improve execution time of network simulators by a factor of 40 and make feasible the 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, precisely defines different variants of staging, and shows how it differs from previous work on reusing computation in simulators. Section 3 shows how intra- and inter-simulation staging can be applied to the core of a wireless network simulator. Section 4 evaluates the performance and scalability benefits of staging in the context of a ubiquitous and mature network simulation engine. Section 5 presents related work, and Section 6 summarizes our contributions.

2. STAGED SIMULATION MODEL

2.1 Formal Simulator Model

We formalize a discrete event simulator as follows. 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, a discrete event simulator also maintains a set of pending *actions*, E_i. In order to compute the next state σ_{i+1}, the simulator choses from E_i the earliest action, which we denote e_i. The simulator transitions from state σ_i to σ_{i+1} according to action e_i, which may modify the state and also schedule or cancel later actions. Although an action e_i may correspond to a simulation-level event, such as a scheduled packet transmission or protocol timer, it can also be used to represent lower level actions of the simulator, such as the sequence of individual function calls or machine-level instructions. We explicitly leave the granularity of these actions unspecified, since our techniques apply to any level for which a cache lookup is faster than reprocessing an action, and use the same notation throughout the paper to describe actions at multiple levels of abstraction.

Formally, we can express an action e_i as a function

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

taking its input from the current state σ_i, and returning 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 action 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 action e_i, the simulator updates the state according to Δ and the pending event set according to Σ. Formally, we write

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

where the function f applies the incremental state change Δ to the current state, and the function g similarly updates the set of remaining actions according to Σ by removing some actions and adding others. For the sake of clarity, we use operators such as union $\cup$ to combine two incremental changes of Δ into a single result.

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, in that they correspond exactly to the output from the original, unoptimized simulator. Yet simulator optimizations aim only to reproduce the operator-visible data from the simulation, and make no attempt to reproduce hidden, implementation-specific details of the simulator's internal operation.

We formalize this notion of simulation equivalence, which ensures that the modified simulator faithfully represents the original simulator, and preserves all useful, operator-visible state. 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 $\forall i, \pi_U(\overline{\sigma}_{\overline{i}}) = \pi_U(\sigma_i)$. Here, $i = \psi(\overline{i})$ is a mapping from the steps of one simulation to those of the other. This mapping is necessary since, in general, 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 action sets E.

This formal description of a discrete event simulator is useful for illustrating and categorizing previous work, as well as for defining staged simulation and exploring the the types of optimizations allowed by our approach. 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

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 actions 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 actions added or removed from the pending action set E_i. Formally, we can describe this second run as a new simulation, beginning in a configuration

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

where the operator selects the value of k and specifies some change Σ to the pending action set. It is never necessary for the operator to modify the state σ_k, since such a change can always be wrapped in an action and placed at the head of the list of pending actions.

The 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 action e_i so long as the relevant parts of the two simulator states $\pi_{e_i}(\sigma_i)$ and $\pi_{e_i}(\sigma'_i)$ are identical. This essentially extends the common prefix technique used in splitting to common prefixes of partial state. Both cloning and splitting typically deal with actions only at the level of simulation events, such as packet transmissions and receptions.

The *updateable simulation* [Ferenci et al. 2002] technique attempts to reuse actions from previous runs of the simulator. An initial run of the simulator caches the Δ and Σ tuples from each action taken by the simulator. Subsequent runs can reuse stored results from previous runs if protocol-specific knowledge dictates that 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 which have not yet been computed.

2.3 Function Caching

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

(1) $e_i = e_j$

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

So, if two actions and their inputs are identical, then the results of one 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 Δ of each action in a table indexed by the inputs to the computation. Before executing an action on new inputs, the table 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 is also applicable between simulation runs if the cache is maintained 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. Typically, almost all simulation state is dependent on the clock value t, which increases monotonically between high-level network events. This is especially true for wireless simulations, where nodes may move. Consequently, it is somewhat rare for two computations to have identical inputs, even between two simulation runs. While function-caching as presented here is strictly more powerful than previous work, its limitations must be addressed if it is to be widely applicable.

2.4 Staged Simulation

Staged simulation improves on function caching by enabling optimization even when inputs are not identical across computations. It does so by relying 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 action. 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 action e_i into two actions

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

where c is the partial result computed by e_i^1 and cached in Δ_1. 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 $y = y'$. In general, the two pieces of the computation need not be performed sequentially, but may be spaced arbitrarily far apart in the sequence of simulator actions. This technique can be applied repeatedly to decompose an expensive computation into a tree of cacheable subcomputations.

Staged simulation employs more advanced techniques for coping with actions with non-identical input as well. It is often possible to restructure computations in order to increase the amount of common computation across actions, then employ caching techniques similar to the one described above. An example of this approach, called *refinement*, is to first compute a *conservative bound* on a result during each computation, then later combine this partial result with more precise information to realize an exact result. Here, the inputs to the auxiliary computation do not need to be identical in order to ensure that the results can be reused between computations. For example, in wireless networks we may be able to find a set of nodes very near to a transmitting node. Later, if the simulator can ensure that the nodes are still sufficiently close, it can avoid computing distances to each of these nodes and instead compute distances for only those nodes which are not included in the previously computed set.

We can formalize this example by transforming an action

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

into two actions

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

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 curring method. Later invocations can reuse results so long as the parameters fall within the limits set on each of the parameters.

We base a third staged simulation technique on *precomputation*. 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 actions simultaneously, rather than individually and spread out over time. Formally, we reorder actions which produce internal results under a mapping ψ, grouping similar computations together in the sequence of computations. For example, if actions a and b have already been decomposed by currying, the simulator might execute the sequence of actions

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

during the simulation. Precomputation can transforms this to

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

if the inputs to b^1 are unchanged by the reordering. This modified sequence may improve efficiency by improving cache locality. For instance, the memory footprint of the simulator exhibits greater locality when all actions for one node are processed in batches at the same time. Moreover, the separate events a^1 and b^1 can typically be combined into a single, more efficient event which computes the results for both computations simultaneously.

3. APPLICATIONS OF STAGING

In this section, we describe how staging can be used to reduce the amount of redundant computation in wireless network simulations and improve their performance and scale. In common wireless simulator implementations, the wireless physical layer and mobility models are the largest consumers of processing time in typical simulation scenarios. These aspects pose the most significant bottlenecks to efficiency and scaling. Consequently, we focus on staging computations related to node mobility and the wireless physical layer.

For staging to be effective, redundant computations need to be readily identifiable, and computations must be restructured and decomposed into independent sub-computations. The monolithic structure of typical implementations, however, obscures the redundant computations performed at runtime. Specifically, simulators must perform numerous calculations to ultimately determine whether a packet will be received by a given node. These calculations depend strongly on the positions of sending and receiving nodes, packet transmission and detection power levels, geography, and radio and antenna models. Thus identical inputs to high level packet related computations are rarely found, either within a single run of a simulator or across multiple similar runs. Further, because of the dynamic nature of wireless networks, nearly all high-level events depend on the clock value t, and thus are unlikely to ever have identical inputs within a single simulation. We therefore look at a lower level and apply staging to individual blocks of code (function calls and subcomputations within functions), rather than to high-level simulation events.

We incrementally describe four different types of staging, 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 actions. The second demonstrates the use of refinement to enlarge the overlap in computation across computations. The third optimization illustrates precomputation as a staging technique, and the final one demonstrates inter-simulation staging by reusing results across multiple similar runs of the simulator.

3.1 Currying: Grid-based Neighborhood Computation

As an initial application of staging, we first restructure the computation that determines the set of nodes within range of a transmitting node. This restructuring is intended to expose the redundancy between and within calls to the neighborhood computation. In the simulator model described in Section 2, we have an action e taking as inputs $\langle t, src\rangle$ and returning $\langle dests, \phi\rangle$. The inputs are the clock value, the source node, and several other parameters not shown, such as the global list of all nodes, a geography, a mobility model describing node movement, and an antenna model describing transmission and reception parameters for each node. The src and $nodes$ inputs specify the x and y coordinates in the geography. The result is a set $dests$ to be returned on the call stack in σ, and an empty set ϕ indicating that this action does not schedule or cancel any other events.

A first example of staging can be seen in straightforward grid-based staging approach, where we reuse previously computed transmission results for nodes which are close by in distance within a single computation, and expose and reuse partial results between packet transmissions. We first divide 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 3.1. This partial information about node positions can then be used to quickly determine if a group of nodes falls entirely outside the

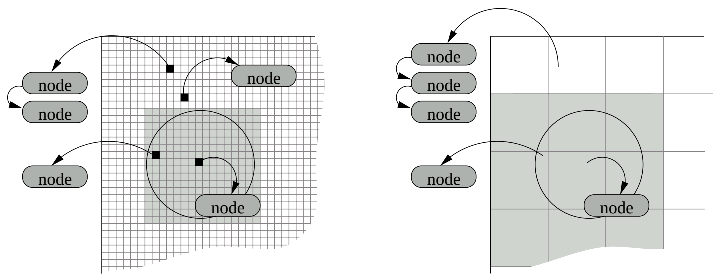

Fig. 1. Grid-based neighborhood computation data structure with either fine or coarse-grained grid granularity. The source node at center is transmitting with radius shown. Examined grid bins are shaded, and linked lists indicated by arrows.

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 $\pi_U(\sigma)$. In general, this will be true of all of our staging techniques.

Grid-based neighborhood computation uses staging in two distinct ways. First, the 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.

3.2 Refinement: Neighborhood Caching

Variations on the grid approach allow more advanced applications of staging using auxiliary computations to reduce redundancy in computation across packet transmissions. In typical 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 neighborhood computation for a node to be reusable across many packet transmissions.

Since inputs will vary slightly, we should not expect the neighborhood set to be identical to that computed during the previous packet transmission. However, a conservative upper-bound, or superset, of the neighborhood set will remain valid for some time after it is computed, depending on the amount of node mobility and the tightness of the bound. We therefore restructure the neighborhood set computation to first compute an upper-bound on the result, then refine this bound into an exact result. After restructuring the computation, intra-simulation staging is used to cache and reuse the common intermediate results, the upper-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

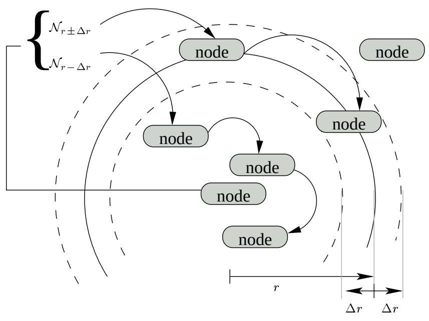

Fig. 2. Neighborhood Cache Entry Structure for Node at Center, Showing Transmission Radius r, $\Delta r = 2s_{max}\Delta t$, and Linked Lists (Arrows) for Sets $\mathcal{N}_{r-\Delta r}$ and $\mathcal{N}_{r\pm\Delta r}$

scenario, then the maximum change in distance between two nodes in an epoch is just $\Delta r = 2s_{max}\Delta t$. 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. Similarly, nodes beyond distance $r + \Delta r$ need not be considered at all for Δt seconds into the future.

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 3.2, 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 mobility (grid) manager 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 analyze these overheads in Section 4.

In terms of the simulator model, this type of staging exposes and leverages redundant computation only in 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. Further, as presented here, caching is only applied when both computations occur within the same simulation run.

3.3 Precomputation: Perfect Caching

We can expand the overlap in computation by looking for other redundancies than that addressed by neighborhood caching. 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 simultaneously, even though the source nodes differ since, in general, each pair of nodes will be considered twice. We use precomputation to address this redundancy, 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 hit the cache. There are several possible advantages to precomputation. First, we only need to examine each pair of nodes at most once, rather than twice, to compute all of the entries. Second, the positions of all nodes can be updated a single time at the start of the generation process. Previously, it was necessary to update the positions of all nodes within range of the sender during each cache miss. Finally, memory locality should improve when precomputing all entries simultaneously as compared to individually on-demand.

The overhead of this technique is a scheduled event during each Δt epoch, and possibly some 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 problem can be addressed directly by appropriately choosing the Δt epoch parameter.

3.4 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. The intra-simulation examples above reduce the amount of computation significantly, but also add some additional events to the event queue leading to more work in the event scheduler. Event queue management is a well-studied problem, especially in the particular case of the Calendar Queue used in ns2. However, we can eliminate the work done by many events by looking at a set of simulation runs together. This application of inter-simulation staging therefore builds on the previous optimizations 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.

First, note that by itself, perfect caching generates strictly more events than the on-demand caching approach, and may actually compute results that are not used in the particular simulation run. But, also observe that perfect caching will perform identical work during multiple simulation runs using the same mobility pattern. In the second and subsequent runs of the simulator we can eliminate these extra events, as well as most cache maintenance, by writing all cache entries to disk every Δt seconds during the first simulator run. Subsequent runs can obtain cache entries from disk rather than maintaining an underlying cache manager or grid. This technique then introduces two phases. The *generation-phase* is identical to perfect caching except that all cache entries are spooled to

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	$\approx$ 20 nodes / km^2
Simulation	
Routing protocol	AODV
Simulation time	400 s

disk. The *use-phase* does not maintain a grid, does not need to track changes to node positions, and requires no scheduler events. Instead, cache entries are read from disk serially as needed during packet transmission. A set of runs with the same mobility model will use the more expensive generation-phase for the first run, and the less expensive use-phase for all remaining runs.

4. EVALUATION

We have implemented each of the optimizations described in Section 3 in the ns2 simulator. 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 more advanced intra-simulation techniques improve stability and robustness of the simulator, while the application of inter-simulation staging improves performance yet further. With the latest staged implementation, we regularly simulate networks of over 1000 nodes in the time it previously took to simulate networks of tens of nodes.

In addition to evaluating total simulation run time using our techniques, we also characterize the effect of each parameter we have introduced. It must be 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. We first describe our test environment and changes required to add staging to the simulator, then present the results of our staging techniques.

4.1 Evaluation Platform and Environment

We take as our baseline the ns2 version 2.1b9a simulator. All 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. Before implementing our staging techniques, we made a few non-standard modifications to improve the baseline ns2 code. Most notably, we disabled all unused packet headers to improve memory locality, and implemented more efficient packet tracing. This improved run time by approximately 85% for all implementations using a 250 node network, including the baseline simulator.

Overall, our simulation runs closely resemble those discussed in [Broch et al. 1998], a very common setup. We used standard CMU Monarch mobility and communication model

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 + Caching
L_3	L_2 + Perfect caching
	Inter-simulation staging
L_{4a}	L_3 + On-disk caching (generation)
L_{4b}	L_3 + On-disk caching (use)

generators from the standard ns2 distribution. As an exemplar of typical wireless network research we chose the AODV ad hoc routing protocol implementation, also included with ns2. Our results are not specific to these choices of application, mobility model, or communication pattern. These system parameters, summarized in Table 4.1, closely follow the standard values used in ad hoc networking literature.

4.2 Simulator Performance

We first examine how the different applications of staging affect total simulation execution time using a 250 node network. 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. We run our simulations with various applications of staging enabled, as shown in Table 4.2.

The speedup achieved by increasing levels of staging relative to the baseline simulator is shown in Figure 4.2. These results, obtained using a 500 node network, highlight especially the benefits of the simplest intra-simulation staging L_1 technique and of the inter-simulation staging $L_{4a} + L_{4b}$ technique. Optimization level L_{4b}, the second phase inter-simulation staging approach, improves simulation run time significantly in comparison to using only intra-simulation techniques. Also, the one-time cost of the first phase, L_{4a} is no worse than the best possible intra-simulation technique L_3. Thus, in this case inter-simulation staging imposes no additional cost during the first run of a series, but offers a significant speedup during subsequent runs.

4.3 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 4.3 shows that staging can improve the scalability of wireless simulators by reducing redundant computations. While not shown on the graph, we have performed basic simulation runs as large as ten thousand nodes in about 40 minutes under the load conditions described before. This level of performance enables the examination of wireless networks on a scale not possible using traditional simulation tools.

This experiment also demonstrates the benefits of inter-simulation staging, which achieves a 10% improvement over the intra-simulation staging techniques. Although the different intra-simulation staging approaches show similar performance in this experiment, they 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 4.3.

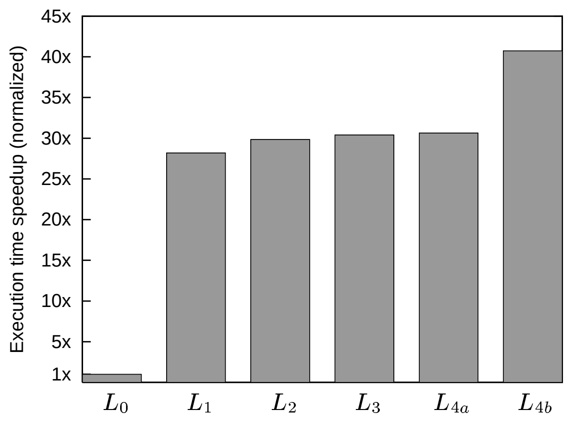

Fig. 3. Speedup in Execution Time with Increasing Staging Relative to Baseline Ns2 Implementation using a 500 Node Network

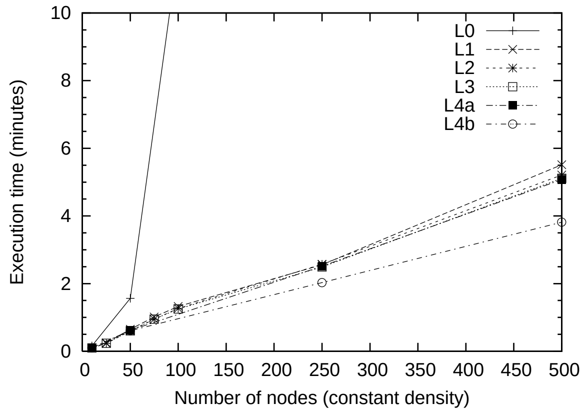

Fig. 4. Effect of Network Size on Total Simulation Run Time Holding Node Density Constant

4.4 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 introduces a granularity parameter, and the caching intra-simulation approach a Δt lookahead parameter.

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 grid-crossing events as nodes move about in the topology, and also leads to more work in packet transmission, as many empty bins will be scanned for nodes. Conversely, a very coarse granularity reduces to a single bucket and, essentially, a scan over all nodes during each packet transmission or cache miss. A reasonable choice is to use the node transmission radius, which requires

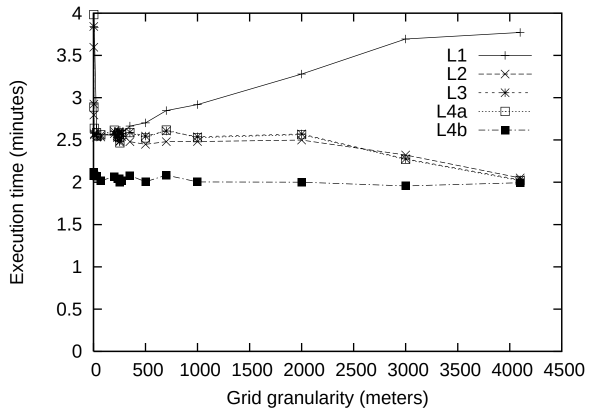

Fig. 5. Effect of Varying Grid Granularity on Simulation Run Time

a scan of roughly nine buckets during each transmission or cache miss.

We run the simulator with the same configuration as before with Δt fixed at 2 seconds, but vary the grid granularity. Figure 4.4 verifies our intuitive description of the effects of grid granularity. Interestingly, we find that any choice of granularity other than the two extremes yields a substantial improvement in run time under L_1 staging, with only minor variation between 100 and 500 meters.

In this experiment, even the right-most extreme performs much better than the ns2 baseline implementation since we avoid creating events and copies of the packet for nodes outside the transmission range. Further, 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 poorly-tuned grid is consulted only in the rare case of a cache miss. Also, the caching techniques L_2, L_3, and L_{4a} appear to improve with a very coarse grid. This is because the cost of maintaining a finer grid is worse than the cost of scanning all 250 nodes during each (rare) cache miss. This would not be the case for much larger networks.

4.5 Caching Lookahead Parameter

The accuracy and overhead of constructing cache entries is controlled by the Δt parameter to the neighborhood caching routine. 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, leading to larger and more imprecise cache values, but allowing 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 4.5 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_3 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

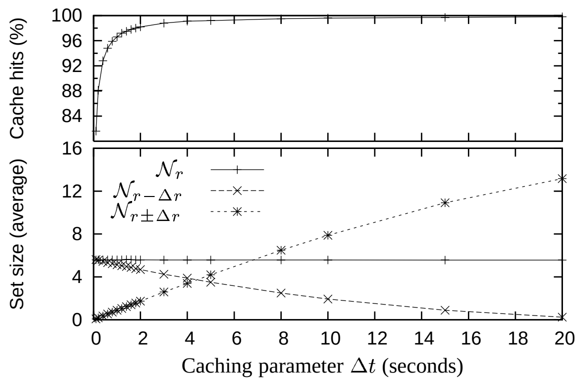

Fig. 6. Effect of Varying Caching Parameter Δt on Cache Hit Rate and Neighborhood Sizes

$\mathcal{N}_{r\pm\Delta r}$. 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. Further experiments indicate that over the entire range of values in Figure 4.5, run time increases by at most 10% as Δt increases. The L_2, L_3, and L_{4a} staging levels all perform similarly, while the second phase L_{4b} inter-simulation approach improves run time by approximately 18% as compared to L_3, independent of the Δt parameter. As with the grid granularity, nearly any reasonable choice of parameter will work well for the highest levels of staging.

5. 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 or recognized in the literature. There has been no prior recognition or development of the technique of staging as a general approach to simulation optimization.

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.

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 particularly inefficient as 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 which 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.

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 to 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], are 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 alter the final result of computations in any way.

Finally, we note that staging has been employed in other settings, most notably in compilers and iterative programming, to achieve performance. Chambers [2002] discusses a staged compilation technique that combines partial evaluation with runtime constant propagation. This technique precompiles code in an initial stage, and fills any holes left by the precompiler with runtime constants at run time, when their values become known. Iterative programming is another area where the general concept of staging has been used. Like staged simulation, iterative programming relies on reusing results, intermediate values, and extraneous values from previous iterations. Liu et al. [1996] discuss methods for automatically extracting this information using program and data-flow analysis. We find this particular approach unsuitable for large and complex simulator implementations, where data-flow and simulation behavior depend very heavily on the particulars of a simulation run. Additionally, the use of multiple languages compounds the difficulty of low-level automatic program analysis.

6. CONCLUSIONS

We propose a general technique, termed staged simulation, for reducing the run time and increasing the scalability of discrete event simulators. The central idea behind staging is to eliminate redundant or partially-redundant computations typically encountered in simulations. Staging relies on precomputing, caching and reusing partial results to eliminate redundancy. We introduce three different techniques, called currying, refinement and precomputation, for exposing and isolating redundant computations within a single run as well as across multiple runs of a discrete-event simulator. 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.

This paper also introduced a formal model of operation for discrete-event simulators. We use these models to characterize previous work, and to derive a precise notion of simulation equivalency, which we then use to illustrate the operation of staging techniques.

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. Intra-simulation optimizations reduce simulator run time by a factor of 30 and improve simulator scalability from networks of hundreds of nodes to networks of ten thousand nodes. An application of inter-simulation staging can reduce run time even further to a factor of 40 over the original non-staged implementation. 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

BOUKERCHE, A., DAS, S., FABBRI, A., AND YILDIZ, O. 1999. Exploiting model independence for parallel PCS network simulation. In *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 *Mobile Computing and Networking*. 85–97.

CHAMBERS, C. 2002. Staged compilation. *ACM SIGPLAN 37,* 3. Invited talk.

FERENCI, S., FUJIMOTO, R., AMMAR, M., PERUMULLA, K., AND RILEY, G. 2002. Updateable network simulations. In *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 *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 *Winter Simulation Conference*.

HUANG, P., ESTRIN, D., AND HEIDEMANN, J. 1998. Enabling large-scale simulations: Selective abstraction approach to the study of multicast protocols. In *MASCOTS*. 241–248.

HYBINETTE, M. AND FUJIMOTO, R. 1997. Cloning: a novel method for interactive parallel simulation. In *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 *European Simulation Interoperability Workshop*.

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

RILEY, G. F., AMMAR, M. H., AND FUJIMOTO, R. 2000. Stateless routing in network simulations. In *MASCOTS*. 524–531.

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

WU, S. AND BONNET, C. 2002. An alternative packet transmission procedure for mobile network simulation. In *SPECTS*.

ZENG, X., BAGRODIA, R., AND GERLA, M. 1998. GloMoSim: A library for parallel simulation of large-scale wireless networks. In *PADS*. 154–161.

Received Month Year; revised Month Year; accepted Month Year