Modeling Multiprocessor Cache Protocol Impact on MPI Performance

Ghassan Chehaibar*, Meriem Zidouni*†
*Bull SAS – Platforms Hardware R&D
Les Clayes Sous Bois, France
Email: {ghassan.chehaibar, meriem.zidouni}@bull.net

Radu Mateescu†
†INRIA / VASY project-team
Saint Ismier, France
Email: radu.mateescu@inria.fr

Abstract

This paper presents a modeling method particularly suited to analyze interactions between Message Passing Interface MPI *library execution and distributed cache coherence protocol. The method is applied to the Ping-Pong benchmark. In addition to overall performance figures like message exchange latency, it also provides detailed analysis elements such as cache miss counts per variable. It is based on formal modeling where functional aspects and performance aspects are integrated by composition and can be refined independently. A key modeling point is that the cache coherence protocol implies that the duration of an access to a variable is not static but is state-dependent. Our Ping-Pong model allows comparison of different primitive implementations in the context of different cache coherence protocols. We applied this approach using Interactive Markov Chain theory and its implementation in the* CADP *toolbox.*

1. Introduction

The range of high-end servers designed and manufactured by BULL includes multiprocessor systems dedicated to high-performance computing (HPC), providing an implementation of the *Message Passing Interface* (MPI) library [1]. Such an HPC system is a network of multiprocessors, each implementing a *Cache-Coherent Distributed Shared Memory* (CC-DSM) architecture. In the message-passing paradigm, processes do not share data but communicate and synchronize by means of MPI primitives. The data of these primitives are shared between processes, which implies that the performance of the library is impacted by cache coherence protocol behavior.

The work presented in this paper is motivated by our need to understand the impact of cache coherence protocol on MPI library performance in order to optimize the library implementation. We are interested not only in the overall performance given by a benchmark execution, we also look for analysis elements such as individual variable miss counts.

Section 2 details the motivation of this research and explains why, instead of simulation, we have chosen formal modeling using the CADP toolbox that supports *Interactive Markov Chains* (IMC) theory. Section 3 describes the modeling method through its application to the Ping-Pong benchmark. We model a software layer comprising the benchmark and MPI primitives algorithms on top of a hardware layer comprising the topological mapping of processes and cache coherence protocol. The main aspect a model should capture is that, at a given time, the latency of an access done by one process depends on the distributed states of all other process caches. The software, the hardware, and the performance aspects are integrated using composition. Section 4 analyzes the results of the Ping-Pong performance evaluation obtained by varying three parameters: processor mapping onto architecture, primitive algorithms and cache coherence protocol. Section 5 concludes this paper and gives future work directions.

2. Motivation and related work

MPI optimization is an extensively investigated domain along various directions: optimizing implementation on a generic architecture like a cluster [2], [3] or on a specific machine [4], [5]; or building configurable benchmarking tools [6]. All this work relies on *measuring* MPI benchmark performance on actual machines (already manufactured hardware), so that one can show that one implementation is more efficient than another. However, when we want to optimize an implementation for a CC-DSM architecture, we need to analyze its interaction with the cache coherence protocol in order to know how many misses occur and on which variables. This information either cannot be obtained with measurement or requires complex instrumentation. Another drawback of measurement is that optimization cannot be done during hardware development phase.

The goal of our research is to build a method particularly suited to analyzing interaction between MPI library execution and distributed cache coherence protocol. This method should allow our software teams to compare and analyze different implementations of the MPI library, providing overall performance figures and analysis elements such as cache miss counts per variable. We are not aware of any published work on this particular subject.

The process of arriving at an adequate performance model of a system is often considered an art [7] in itself. In our context, this is even more critical, because the performance properties depend heavily on the correct modeling of the functionality of the cache coherence protocol. Therefore, functional modeling and performance modeling need to be very tightly intertwined, and it is useful to have checkers to verify the correctness of the functional behavior at hand. Standard simulation environments such as Opnet (http://www.opnet.com), Omnet (http://www.omnetpp.org) or NS-2 (http://www.isi.edu/nsnam/ns), do not provide any support for this. Instead, in these environments, the model is coded in a loosely structured set of declarations and libraries, and often some C-code snippets are used to glue the components together. The results obtained by such studies are often not credible, as reported in [8], [9], [10]. This is not a deficiency of the analysis technique (discrete event simulation), but the model construction process is badly supported, and hence one looses the view of what one is actually modeling (and thus analyzing).

In this paper we instead use a formal method, the ISO-standardized language LOTOS [11], to model the functional aspects of our system. We formally verify the correctness of this model exploiting model checking capabilities of the tool CADP [12]. On top of that, we integrate performance aspects of the system using the same tool and technique [13], based on the IMC theory [14], which is a smooth extension of the theory underlying LOTOS. This modeling results in a performance model, a highly unstructured continuous time Markov chain, which we are confident properly reflects the functionality and the performance aspects of the system under study. We then use numerical analysis algorithms to calculate relevant performance figures. We could also use discrete-event simulation of a formal model as an alternative analysis technique, but this is currently not supported by CADP. The tight integration of performance and functionality considerations was first devised in the context of stochastic Petri nets [15], and indeed the use of a Petri-net based tool would have been an alternative; we are however not aware of a tool with the same model checking support as CADP provides.

3. Functional and performance modeling

This section describes the modeling process: first we translate the software program into a LOTOS model, then we compose this model with a process that represents the cache protocol and associates to each access its state-dependent transfer type. Finally we obtain a performance model by composition with a process that inserts "delay transitions".

3.1. Ping-Pong functional behavior

In this paper, we consider the MPI benchmark called *Ping-Pong* dedicated to measure the latency of a message transfer between two processes. It is composed of two processes; each process alternately sends and receives a message from the other. If $\mathrm{Snd}_{i\to j}$ denotes the emission of a message from P_i to P_j and $\mathrm{Rcv}_{i\leftarrow j}$ denotes the reception by P_i of a message sent by P_j, then Ping-Pong behavior is: $(\mathrm{Snd}_{0\to 1};\mathrm{Rcv}_{0\leftarrow 1})^k||(\mathrm{Rcv}_{1\leftarrow 0};\mathrm{Snd}_{1\to 0})^k$.

The delay Δt of executing the loop k times is measured; since one loop turn corresponds to a round trip, that is two transfers, one transfer latency l is obtained by $l = \frac{\Delta t}{2k}$.

Each MPI *send* and *receive* primitive is a C procedure composed of assignments, tests, and loops accessing some variables in shared memory. From a functional viewpoint, the fact that memory is physically distributed and that an access to a variable may involve a transfer over the network is not relevant: only the values of variables are of interest. That is why we write a process Memory managing a storage Mem containing the values of variables, that can be accessed atomically in one transition. An access to a variable can be a load or a store and is performed by rendezvous with Memory process on a gate of the form:

```
Action ?P:Proc ?Op:Access ?Var:Address ?Val:Value;
```

where P is the identifier of the process performing the access, Op is the access type (load, store, test-and-set), Var is the name of the variable and Val is the value read or stored.

Each primitive algorithm is composed of *assignments*, *if-then-else*, and *while* constructs. A straightforward translation of these instructions into Action gates gives two LOTOS procedures Send(Pi, Pj) and Receive(Pi, Pj). Then, each process of Ping-Pong is:

Proc0[Action] = Send(P0, P1);Receive (P0, P1);Proc0 and

Proc1[Action] = Receive(P1, P0);Send(P1, P0);Proc1,

and the functional behavior of the benchmark is:

```
(Proc0[Action] ||| Proc1[Action]) |[Action]| Memory[Action](Mem)
```

(";" is the sequential composition of processes, ||| is the free interleaving and |[A]| is synchronization on action A).

3.2. Cache coherence protocol impact

Our protocol is based on the classic 4-state protocol called MESI [16] (acronym formed by the state initials): Modified, Exclusive, Shared, and Invalid. The cache coherence protocol is described in Table 1 for a load access and Table 2 for a store access. P_i is the requesting processor and P_j represents all other processors. The first two columns give the current cache states, and the next two ones give the new cache states after the data transfer. The fifth column gives the data transfer type, specifying the data provider (M means home memory) and occasional cache lookup or invalidation. We consider two protocol variants that differ only in handling a *store* miss when modified data exist (shown by the last two lines): in case A the requester goes into *M* state and the provider into *I* state; in case B they both go into *S* state.

These tables define a cache state transition function *NewState(p, op, v, CurrentState)* where *p* is the requesting processor, *op* is the access performed (load or store) and *v* the variable that is accessed. It is worth noting that cache

Table 1. Protocol for a load operation

CurrentState		NewState		Data transfer type
P_i	P_j	P_i	P_j	
S/E/M	*	S/E/M	*	Hit (internal)
I	I	E	I	$M \rightarrow P_i$
I	S	S	S	$M \rightarrow P_i$
I	E	S	S	$M \rightarrow P_i$ with P_j lookup
I	I	E	I	$P_j \rightarrow P_i$

Table 2. Protocol for a store operation

CurrentState		NewState		Data transfer type
P_i	P_j	P_i	P_j	
E/M	I	E/M	I	Hit (internal)
I/S	I	M	I	$M \rightarrow P_i$
I/S	S	M	I	$M \rightarrow P_i$ with P_j invalidation
I	E	M	I	$M \rightarrow P_i$ with P_j invalidation
I	M	M	I	$P_j \rightarrow P_i$ (Protocol A)
I	M	S	S	$P_j \rightarrow P_i$ (Protocol B)

states in the real system are not modified atomically but the protocol should ensure that from a coherence viewpoint, the system behaves as though the *NewState* function is atomic. The tables also define a *Transfer(p, op, v, CurrentState, vmap, pmap)* function giving the data transfer type, which has two additional parameters *vmap* and *pmap*, describing respectively variable mapping in the distributed memory and process mapping onto processors. We consider the FAME architecture commercialized in the Bull NovaScale® "Intensive Line" servers (www.bull.com/novascale/intensive.php). It is a hierarchical CC-DSM architecture with three levels: at the top level a system is composed of modules, a module is composed of processing nodes, and a processing node contains processors connected with a bus and equipped with a memory piece. This gives three possible cases of distance (denoted d) between a requester and a data provider: in the same node ($d = 0$), in different nodes but in the same module ($d = 1$) and in different modules ($d = 2$). Therefore, *Transfer* function can take 13 possible values: 1 hit case and 12 miss cases. The miss cases are obtained by varying 3 parameters: distance between source and destination (3 values), source types (2 values: other cache or memory), handling in memory source case (3 cases: with other cache lookup, with invalidation, or none of them).

It follows from *NewState* and *Transfer* definitions that the *latency of an access* to a given variable in a given process cannot be determined statically or locally because it does not depend only on its local cache state, or on variable or process mapping onto the architecture. It also depends on the *global state of all caches* in the system.

3.3. Inserting state dependent latencies

Once the functional model is built, the next step is to associate a duration to each access performed by the benchmark. In IMC a transition can be an "ordinary" immediate action, or a delay action labeled with "label; rate R", where *R* is a positive real and *label* any string (possibly empty). A transition $S_1 \rightarrow S_2$ labeled with *rate R* indicates that the probability that this transition occurs at $t \leq x$ is $1 - e^{-Rx}$.

Applying this approach in LOTOS, in order to specify that an action A has a mean duration (or latency) L_A, we write the sequence Begin_A; Latency_A; End_A, where Begin_A and End_A are immediate actions and Latency_A is a delay one. This distinction is not done at the LOTOS level but in the generated labeled transition system (LTS), where Latency_A is renamed with Rate R or Label; Rate R, where R=1/L_A: this gives a so-called stochastic LTS.

We will extend our functional model to introduce access latencies in two steps using process composition: first we associate to each access its transfer type, then we insert delay transitions as explained above. Concerning transfer types, we write a process Caches that maintains the global state of all caches for all variables and gives the transfer type associated to each access (as defined by the function *Transfer* of Section 3.2):

```
Process Caches[Action](Cs:Cachestates, Map:Mapping): Noexit:=
  Action ?P:Proc ?Op:Access ?V:Address ?Val:Value ?T:Transfer
    [T == Transfer(P, Map, Op, V, Cs)];
  Caches[Action](Newstate(P, Op, V, Cs), Map)
Endproc
```

Then we add the ?T:Transfertype on the gates of the functional behavior and we compose it with Caches, thus obtaining a functional model where accesses are labeled with their transfer types. In order to insert delay transitions as explained above, each occurrence of the gate Action in Send and Receive processes has to be split into a sequence Action;End_action (we keep Action instead of Begin_Action). Then the delay transitions are inserted by composition with the following process:

```
Process Latency[Action, End_action, Latency](P:Proc): Noexit :=
  Action !P ?Op:Access ?V:Address ?Val:Value ?T:Transfer;
  Latency !P !V !T;
  End_action !P;
  Latency[Action, End_action, Latency](P)
Endproc
```

yielding the following processes:

```
PLj[Action, End_action, Latency]=
  Procj[Action, End_action]
    |[Action, End_action]|
  Latency[Action, End_action, Latency](Pj)
```

and the benchmark whole model is:

```
Benchmark =
(  PL0[Action, End_action, Latency]
     |||
   PL1[Action, End_action, Latency]
)
     |[Action]|
   Memory[Action](Mem)
     |[Action]|
   Caches[Action](Cs, Map)
```

In Memory and Caches processes, actions are not split because these rendezvous are an abstraction to provide variable values and cache states to the benchmark processes and do not represent actual transfers: no duration can be associated to them in these processes.

The model built above is very abstract, since it associates a one-shot latency to a transfer and it makes cache state transition one atomic event at the beginning of the transfer. In the real system, on a cache miss, the requesting processor issues a transaction that goes through many phases in the various agents of the architecture. Cache states are changed according to these phases. Actually the compositional way of inserting latencies allows us to easily refine the model to choose the needed abstraction level in order to improve accuracy. For instance, in order to split a transfer into two equal phases and put cache state update at the end of the first phase, we modify only processes Caches and Latency. We keep cache states unchanged when Action is fired and we add a specific event Update in these processes as follows:

```
Process Caches[Action](Cs:Cachestates, Map:Mapping): Noexit:=
   Action ?P:Proc ?Op:Access ?V:Address ?Val:Value ?T:Transfer
      [T == Transfer(P, Map, Op, V, Cs)];
   Caches[Action](Cs, Map)
 []
   Update ?P:Proc ?Op:Access ?V:Address;
   Caches[Action](Newstate(P, Op, V, Cs), Map)
Endproc
```

Then Latency process is modified this way:

```
Process Latency[Action, End_action, Latency, Update](P:Proc):
Noexit :=
   Action !P ?Op:Access ?V:Address ?Val:Value ?T:Transfer;
   Latency !P !V !T !1;
   Update !P !Op !V;
   Latency !P !V !T !2;
   End_action !P;
   Latency[Action, End_action, Latency](P)
Endproc
```

This shows how this modeling separates the functional aspect from the performance one: in this way, each aspect can be refined independently.

3.4. Formal verification

Once the LOTOS model is available, before exploiting it for performance evaluation, we assess its correct functioning using the formal verification tools of CADP toolbox based on equivalence checking and model checking. The following verifications are done:

- When all the actions of the model are hidden except the actions that send or receive a message and branching bisimulation is applied, we obtain a behavior equivalent to the Ping-Pong benchmark.
- Correct functioning of the cache coherence protocol is checked using safety properties written in regular alternation-free μ-calculus, the input language of the EVALUATOR [17] model checker of CADP.
- Correct functioning of the primitive algorithms, such as lock management, is also checked using temporal logic properties written in regular alternation-free μ-calculus.

4. Performance analysis

To perform numerical analysis, we generate the LTS of the model, then we rename transitions labeled Latency !P !V !T with P_V_T; rate R where $R=1/L$ if L is the latency of a transfer of type *T*. BCG_MIN minimization tool [18] is used to reduce LTS according to stochastic branching bi-simulation. The BCG_STEADY tool [18] takes a stochastic LTS as input, and computes the corresponding equilibrium ("steady-state") probability distribution on a long run. Moreover, it also computes throughputs of the transitions bearing a given *label*, which is the only output we have used in this study.

4.1. Computing performance figures

In assessing Ping-Pong performance, two figures are of interest: latency of a message exchange, and number of misses on a given variable during this exchange, to analyze the impact of each variable used in the primitives implementation.

The mean latency between two consecutive occurrences of an event is the inverse of its throughput. So, in order to compute the latency of a message exchange in the Ping-Pong benchmark model, we add a Startj gate in the LOTOS specification at the beginning of Procj, and we compute $L = 1/(2{\cdot}F_S)$ where F_S is the throughput of Start (see Section 3.1 for a justification of the factor $1/2$).

In order to obtain the average number of misses on a given variable during a message exchange in Ping-Pong, we notice that a miss on a variable V corresponds to an occurrence of a miss transfer type, i.e. an action Latency !P !V !T where T is a miss case. But the mean number of occurrences of an event in a loop is the throughput of this event divided by the throughput of the occurrence of the loop. Then, if we call $F(V, p)$ the sum of all throughputs of Latency !P !V !T, the number of misses on V during a message transfer in a given process is $F(V, p)/F_S$.

4.2. Evaluation results

We evaluate the latency and cache miss numbers of Ping-Pong in several configurations in order to investigate the impact of various parameters on the performance of MPI library. A configuration is defined by three parameters:

1) A mapping of processors (with an associated variable mapping) onto the architecture. This is defined by inter-process distance (0, 1 or 2).
2) An implementation of *Send* and *Receive* primitives. The first one (called SR1 below), provided by MPICH v1.26 (www-unix.mcs.anl.gov/mpi/mpich1) is based on lists and locks: each process has a list of available packets (to use) and a list of incoming packets (received from the other process). These lists are accessed by locking them first. The second implementation

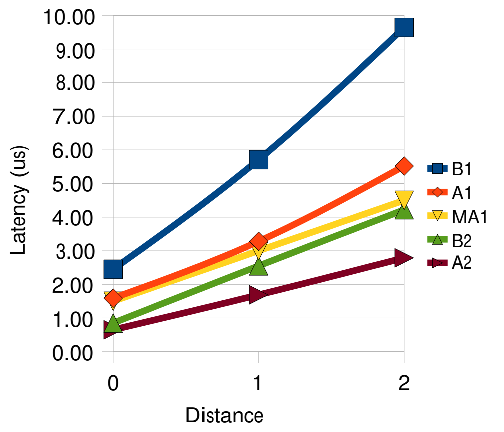

Figure 1. Latency of a message exchange

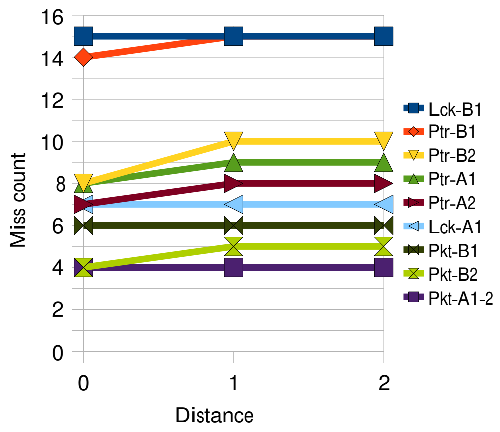

Figure 2. Number of cache misses

(called SR2 below) is a lock-free implementation, which replaces lists with fixed-size buffers, taking advantage of the fact that these buffers have a unique reader (similar to the algorithms described in [4]).

3) A cache coherence protocol: there are two variants (A and B) defined in Section 3.2.

Figure 1 gives the latency results for all cases: Xi curve where X=A or B, and i=1 or 2, represents the X protocol case with SRi algorithm. We also represent the *measured* latency in case A1 (noted MA1) on a Bull server. First we see that in A1 case, the difference between measured and evaluated latency ranges between 9% and 22%. However, let us stress that our objective is not to build a model that accurately predicts absolute experimental values, but to show that a fairly abstract model of a large-scale distributed cache coherence protocol has potential to compare configurations and to show right tendencies. As expected, latency increases when distance between processes increases, in all primitive and protocol cases.

The second less obvious result is that Protocol A is more efficient than Protocol B. Assume a process P_0 performs a load(V) and V is M in P_1 cache (this means that P_1 has already performed a store(V)). If load(V) brings V in E state in P_0 and makes it I in P_1 (Protocol A), any subsequent access by P_0 (load(V) or store(V)) is a hit, and any by P_1 is a miss. Now if load(V) makes V in S state in both caches (Protocol B), a subsequent load(V) is a hit, but store(V) is a miss in both processors. In the MPI library primitive implementation (in both cases SR1 and SR2), a process never performs a store(V) followed by a load(V), which is the favorable case of Protocol B; however, a process always performs store(V) after load(V), which is the favorable case of Protocol A.

The third result is that SR2 implementation is more efficient than SR1 one. This is explained by Figure 2, which gives the number of misses per kind of variables (packet, pointer or lock), in each configuration (Ptr-A1 curve gives the number of misses on pointer variables in A1 case, etc). We see that there are about 20 misses in a message transfer in case A1, where 20% are on packets, 45% on pointers and 35% on locks. In case A2, there are about 12 misses, where 32% are on packets and 68% on pointers (actually buffer index). These figures show that, in SR1, misses on list management variables (pointers and locks) play a major part in the overall latency of a message transfer: therefore a lock-free algorithm like SR2 is more efficient.

4.3. Modeling and evaluation cost

The model size is 797 lines of LOTOS and 1044 lines of C code (for efficiency reasons, data types are implemented in C). The generated LTS has 1,452,856 states. There are 450 lines of temporal logic properties. The state space generation, formal verification and performance evaluation are completed within a few minutes. This shows that in both development effort and execution time, this method is inexpensive and yet yields useful results.

5. Conclusion and future work

In this paper we presented, by means of its application to the Ping-Pong benchmark, a method to analyze the impact of cache coherence protocol on the MPI library performance in a CC-DSM architecture. The goal of this method is to help software teams optimize MPI implementation during hardware design phase and to analyze measurement afterwards. It is based on formal modeling that integrates functional and performance behaviors, using IMC extension of the LOTOS language supported by CADP toolbox. The model's functional behavior is formally verified, which cannot be done in simulation approach. The software aspect (benchmark and MPI library primitive), the hardware aspect (cache protocol and topological mapping), and the performance aspect (delay transitions) are in separate processes. Their integration is done by composition but each aspect can be refined independently. An abstraction of the hardware cache protocol allows to easily capture the fact that an access latency in one process depends on all cache states in the system. This is allowed by the compositional method of building the model and by the distinction of immediate transitions from delay transitions in IMC theory. It computes overall latency and above all cache miss count per variable, in different software and hardware configurations. These detailed figures allow to understand the weight of each access in the overall latency. The model is relatively small and its formal verification and performance evaluation are done within a few minutes. So even such an abstract model of a complex large-scale architecture has the potential to compare and analyze benchmark behavior.

Next step is to apply this method to predict additional figures like bandwidth; and to investigate modeling of other MPI primitives like the barrier one. Our long-term objective is to build a modeling tool that automates this method. It takes in input primitive and benchmark implementations written in some pseudo-code, cache coherence protocol and processor mapping, then automatically produces the LOTOS models and computes the performance figures.

Acknowledgment

We are grateful to Holger Hermanns and Hubert Garavel for useful discussions. This research was supported by the MULTIVAL project of the MINALOGIC "Pôle de compétitivité". M. Zidouni's work is partially financed by French Research Ministry and European Social Fund.

References

[1] M. Snir and et al, *MPI: The Complete Reference (2 volumes)*. MIT Press, 1998.

[2] W. keng Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and S. Tidemann, "Collective caching: Application-aware client-side file caching," in *Proc. HPDC*, 2005.

[3] T. Hoefler, C. Siebert, and W. Rehm, "A practically constant-time MPI broadcast algorithm for large-scale InfiniBand clusters with multicast," in *Proc. IPDPS*. IEEE, 2007, pp. 1–8.

[4] W. Gropp and E. Lusk, "A high-performance MPI implementation on a shared-memory vector supercomputer," *Parallel Computing*, vol. 22, no. 11, pp. 1512–1526, January 1997.

[5] K. Feind and K. McMahon, "An ultrahigh performance MPI implementation on SGI® ccNUMA Altix® systems," *Computational Methods in Science and Technology*, vol. Special issue, pp. 67–70, 2006.

[6] R. Reussner, P. Sanders, L. Prechelt, and M. Müller, "SKaMPI: A detailed, accurate MPI benchmark," in *Proc. 5th European PVM/MPI Users Group Meeting*, ser. LNCS, vol. 1497. Springer Verlag, 1998.

[7] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, 1991.

[8] D. Cavin, Y.Sasson, and A. Schiper, "On the accuracy of manet simulators," in *Proc. 2nd ACM international workshop on Principles of mobile computing*, 2002, pp. 38–43.

[9] G. Riley and M. Ammar, "Simulating large networks: How big is big enough?" in *Proc. First International Conference on Grand Challenges for Modeling and Simulation*, 2002.

[10] J. Heidemann, N. Bulusu, J. Elson, C. Intanagonwiwat, K. Lan, Y. Xu, W. Ye, D. Estrin, and R. Govindan, "Effects of detail in wireless network simulation," in *Proc. SCS Multiconference on Distributed Simulation*, 2001, pp. 3–11.

[11] ISO/IEC, "LOTOS — a formal description technique based on the temporal ordering of observational behaviour," ISO, International Standard 8807, 1989.

[12] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2006: A toolbox for the construction and analysis of distributed processes," in *Proc. CAV*, ser. LNCS, W. Damm and H. Hermanns, Eds., vol. 4590. Springer Verlag, 2007, pp. 158–163.

[13] H. Garavel and H. Hermanns, "On combining functional verification and performance evaluation using CADP," in *Proc. FME*, ser. LNCS, L.-H. Eriksson and P. A. Lindsay, Eds., vol. 2391. Springer Verlag, 2002, pp. 410–429.

[14] H. Hermanns, *Interactive Markov Chains and the Quest for Quantified Quality*, ser. LNCS. Springer Verlag, 2002, vol. 2428.

[15] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.

[16] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution multiprocessor with private cache memories," in *Proc. 11th Annual International Symposium on Computer Architecture*, 1984, pp. 348–354.

[17] R. Mateescu and M. Sighireanu, "Efficient on-the-fly model-checking for regular alternation-free mu-calculus," *Science of Computer Programming*, vol. 46, no. 3, pp. 255–281, March 2003.

[18] H. Hermanns and C. Joubert, "A set of performance and dependability analysis components for CADP," in *Proc. TACAS*, ser. LNCS, H. Garavel and J. Hatcliff, Eds., vol. 2619. Springer Verlag, 2003, pp. 425–430.