The "Neocortex" Neural Simulator
A Modern Design

Raul C. Mureşan
1. Nivis Research
Gh. Bilaşcu 85, 3400 Cluj-Napoca
2. Faculty of Computer Science and Automation
Technical University Cluj-Napoca
Gh. Barițiu 26-28, 3400 Cluj-Napoca
Romania
raulmuresan@yahoo.com

Iosif Ignat
Faculty of Computer Science and Automation
Technical University Cluj-Napoca
Gh. Barițiu 26-28, 3400 Cluj-Napoca
Romania
iosif.ignat@cs.utcluj.ro

***Abstract* – Among the latest developments in simulation techniques, the "Neocortex" simulation environment is based of modern design principles. We present the most important abstractions and building blocks of the system and describing the solutions that overcome many computational challenges of neural simulators. Also, we refer to the scalability of the simulator during the tracing of large neural architectures with different levels of modeling detail.**

I. INTRODUCTION

As it has been previously emphasized [8, 9], neural simulators face many challenging problems, especially when the simulated models are large, containing thousands of neurons and millions of synapses.

When designing a neural simulator, we should first understand the problems it has to deal with, when the complexity of the model is really high (usually when the model / the detail level of the electrophysiological model is large / high). The main aspects to take into account are [9]:

- the **speed** of simulation;
- the required **memory**;
- the **accuracy** of the simulation.

Although these aspects are usually in conflict (eg.: increasing the accuracy decreases the speed), tradeoffs can be found in many cases. The possibility of configuring the simulator for these tradeoffs depends on its **flexibility**. Interfacing of different units with different modeling detail, using different simulation techniques like event-driven and iterative, at the same time, can greatly improve the simulation performance.

The most recent neural simulators either lack generality, like "SpikeNET" or "RetinotopicNET" [3, 8] or they lack flexibility, like "Neuron" or "Genesis" [1]. In the former category, the design of the simulator is committed to some particular type of neural models, a solution which seems to greatly improve the speed of simulation and reduce the memory consumption. On the other hand, simulators in the latter group, are more general, allowing for many levels of detail for the neural models. It turns out that this very generality slows down the entire simulation and increases the required memory. Only small networks of neurons are fit for these simulators. A flexible neural simulator could take advantage of some particular aspects for modeling the large populations where detail is not an issue and, at the same time, it could interface these modules with more detailed ones, where accuracy is critical for the study of neural dynamics. Such an interface could be possible if:

- there is **data compatibility** between the different representations of the models;
- we implement **processing compatibility**, which means that event-driven and iterative processes can synchronize and exchange data.

Having made these observations, we set out to build a novel generation of simulators, starting with "Neocortex".

II. METHODS

We tried to build a neural simulator that takes advantage of modern design techniques and maximizes flexibility. "Neocortex" is based on the following principles:

1. it is able to model different types of spiking neurons, with different detailing;
2. static and dynamic synapses, with or without plasticity, with temporal or a-temporal dynamics, can be implemented;
3. neurons can be grouped together using some criteria, and there are group management functions implemented;
4. the required simulation memory is minimized by avoiding redundant data representation, synaptic sharing, and the extensive use of object association;
5. both implicit and explicit synaptic connections can be implemented;
6. the simulation can be event-driven for deterministic analytical models, or iterative;
7. event-driven and iterative simulations can coexist, data exchange between modules being implemented in the simulator;
8. class inheritance is avoided and inline function expansion is used to a maximum extent;
9. flexibility is maximized by allowing different models, simulation techniques and modularization levels to coexist at the same time.

A. Data representation - modeling

"Neocortex" is written in ANSI C with object C++ extensions. The entire design is object-oriented, however generalization and specialization being replaced by association.

The most important classes are the *SpikingNeuron* and the *SpikingSynapse*. Each of these classes store only a minimum amount of information, which relates to the dynamical state of the entity. For example, the *SpikingNeuron* stores only the membrane potential, a

recovery variable (used in reduced bi-dimensional models) and a spike triggering state (which indicates the presence of a spike during the present time slice).

Apart from the dynamical state of the entity, each instance of the *SpikingNeuron* or *SpikingSynapse* maintains a pointer to a so-called "*Property Class*" object. For neurons we have the *SpikingNeuronClass* and for synapses, the *SpikingSynapseClass*. Each of these property classes store the exact parameters of a whole range of neuron or synapse types. As an example, we might have a "RegularSpikingNeuron" object which specifies the electrophysiological parameters of every regular spiking neuron in the model. Independent of the number of neurons, we always have only one property object for a given neural type, that describes the information shared by all the neurons of the given type. In this way, redundant information is concentrated into one single representation and memory consumption is greatly reduced. We make sure that each neuron and each synapse use only the minimum amount of memory that is required for representing its dynamical state.

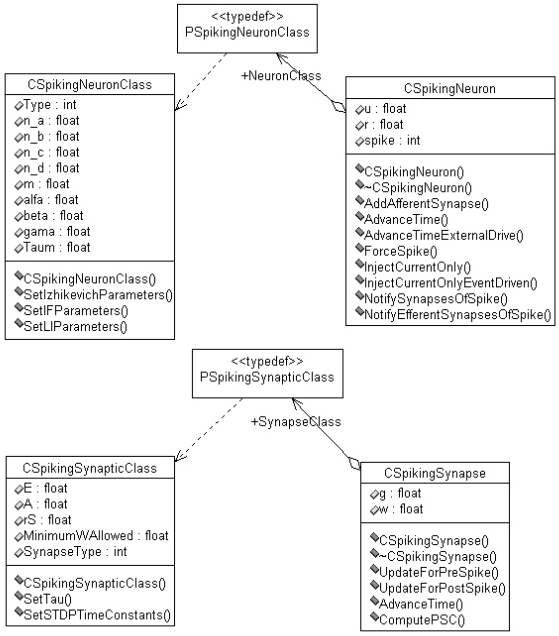

Fig. 1. The *SpikingNeuron* and *SpikingSynapse* classes and the associated property classes which represent the shared information among a class of neurons or synapses (eg. regular spiking neurons or STDP synapses).

In the present implementation, the researcher has 3 available neural models to work with: the linear additive integrating neuron (with no leak and no refractory period [8]), the leaky Integrate-and-Fire (IF) neuron model (see Dayan and Abbot [2]), the model of Izhikevich [5]. The integrative models are uni-dimensional and use only the membrane potential to represent the dynamical state of the neuron. The model of Izhikevich, is a reduced bi-dimensional set of differential equations that is integrated with 1 ms time step, using the Euler method.

The synaptic modeling allows for the implementation of static, dynamic and STDP synapses. In the case of static synapses, a given, fixed current is injected as a delta pulse into the target neuron as in (1) for each incoming stimulation.

$$PSC(t) = \begin{cases} w, if\ t = t_{in\_spike} \\ 0, otherwise \end{cases}, \quad (1)$$

where *PSC* is the postsynaptic current, t_{in_spike} is the moment of presynaptic spike and *w* is the weight of the synapse.

For dynamical synapses, the PSC is time dependent and is implemented as an alfa function [4]. Each time there is a presynaptic event, the conductance *g*, is incremented by 1. At the same time, the value of *g* is exponentially decaying with a given time constant.

$$\begin{aligned} &if\ (presynaptic\_spike)\ then\ g(t) = g(t) + 1 \\ &g(t) = g(t) * e^{-\frac{1}{\tau}} \quad (dt\ is\ 1\ ms) \\ &PSC(t) = A * w * g(t) * (E_{syn} - u(t)) \end{aligned}, \quad (2)$$

where *g* is the time dependent conductance, *A* is the maximum PSC amplitude, *w* is the synapse weight (between 0..1), E_{syn} is the reversal potential for the synapse (typically 0 mV for excitatory and -90 mV for inhibitory synapses), *u* is the time dependent membrane potential of the postsynaptic neuron.

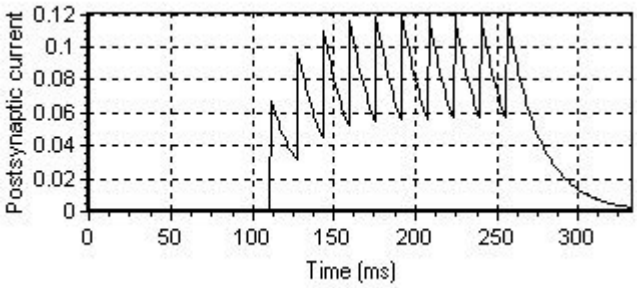

Fig. 2. The postsynaptic current of a dynamic synapse. The presynaptic stimulation is generated by a 60 Hz spiking neuron.

For the spike-timing dependent (STDP) synapses, the synaptic weight *w* is not fixed any more but time dependent *w(t)*. The time dependent plasticity algorithm implemented here is based on the synaptic release probability, inspired by the model of Senn and Markram [10].

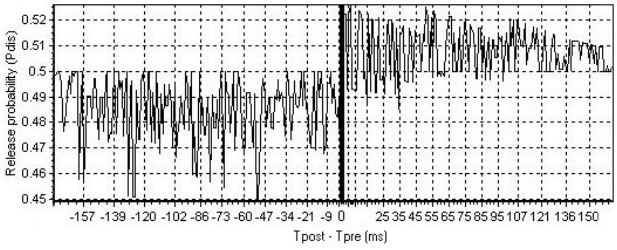

Fig. 3. Synaptic release probability for a STDP synapse

The dynamics of the STDP variables are computed using an additional associated object, called *STDP-Variables*. Only STDP synapses have an associated STDP variable object, so that the representation of the generic synapse is as compact as possible. When the type of the synapse is STDP, the simulator uses the *STDPVariables* object to compute the value of the synaptic weight *w(t)*. At the same time, it computes the new values for the STDP variables.

B. Neural maps

Neuron and synapse objects can be created and managed by the user. However, there is an additional abstraction layer that can be used to formalize better the concept of neuronal group (we refer here to neural grouping as a physical segregation process). As in the neocortex, neurons can be grouped together according to some criteria like: receptive field properties, cortical layer and spatial distribution, connectivity, etc. Usually, neighboring neurons are more likely to be involved in the same type of processing so it is a good idea to group them together. A group of similar neurons is called "neural map".

The "Neocortex" simulator allows the user to create neural maps, which are two-dimensional layers of neurons that can have external connections (with other maps) and/or lateral connections (within the current map).

Each neuron in a map is uniquely identified by its cartesian spatial position *(x,y)* (Fig. 4).

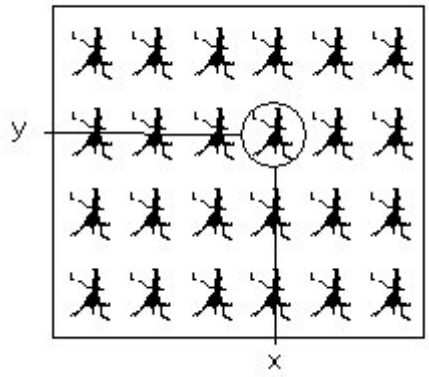

Fig. 4. The matrix-like structure of a neural map

Using neural maps is especially useful when modeling hierarchical systems like the visual cortex [7]. At the level of the neural map, many management functions can be implemented, like cleanup, reset, etc.

There is an additional advantage in using maps to group neurons. When connections between maps are retinotopic and static, one can take advantage of synaptic sharing. The user could then define a rule of connectivity between maps, which is called implicit synapse representation. Implicit representations of synapses are also called synaptic kernels and are usually matrixes that contain the spatial distribution of the synapse weights (Fig. 5).

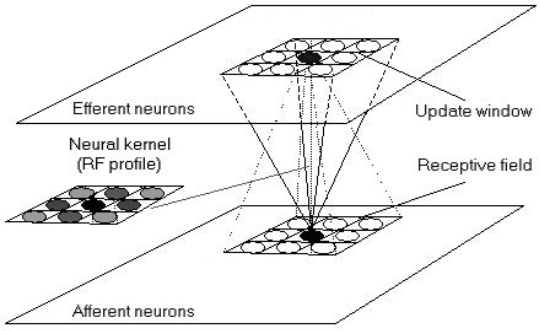

Fig. 5. Implicit synapses between two retinotopically connected maps

Whenever the value of a synapse is required, a correspondence algorithm is applied, which takes the position of the afferent and efferent neurons as an input and determines the exact value of the synapse from the synaptic kernel [8].

We have to mention that implicit synapse representation can only be used when:

- all the neurons in a given map share the same receptive field profile;
- the receptive fields of the target map neurons are retinotopic like;
- the user is not interested in dynamical synapses (between the two maps) and a simple additive PSC is sufficient for simulation.

Maps can be configured to automatically generate implicit synaptic kernels based on given kernel functions. At the same time, when the user is interested in using more elaborate models of synapses, with dynamic states, the simulator can automatically generate explicit synapses based on a kernel function.

$$K_{C-S}(x,y) = onGain \cdot e^{\frac{-(x^2+y^2)}{2\cdot\sigma_1}} + offGain \cdot e^{\frac{-(x^2+y^2)}{2\cdot\sigma_2}} \quad (3)$$

In (3) we present an example of center-surround gaussian kernel function. Depending on the gains (*on/offGain*) and the standard deviations ($\sigma_{1,2}$), mexican-hat or other profiles can be obtained.

C. Simulation techniques

Most of the simulators used today are based on iterative simulation. The user defines a time step, which can range from sub-milliseconds to tens of seconds. Depending on the models that are used, the time step should be chosen appropriately so that the integration of the differential equations minimizes the error and maximizes the integration accuracy. However, choosing too small time steps might not significantly increase the accuracy while it increases the simulation time for sure.

During iterative simulation, the state of each dynamical variable is updated for every time step. So to speak, the present state of each neuron / synapse / variable is computed from the previous state and it is used to compute the next state. An important observation if that for simplified models, like the linear additive or the leaky integrate-and-fire neuron, there is deterministic behavior between time steps, given that no PSC is injected in the neuron during this period. In these cases, the state of the neuron can be computed with 100% reliability from any previous state, if the neuron hasn't been stimulated since.

In (4) you'll find the update formulas for the linear additive (LA) and leaky IF neurons, computed from a previous state (t_p). This only holds if the neuron hasn't been stimulated since t_p.

$$U_{LA}(t) = U_{LA}(t_p)$$
$$U_{IF}(t) = U_R + (U_{IF}(t_p) - U_R) * e^{-\frac{(t-t_p)}{\tau}}, \quad (4)$$

where U_{LA} is the membrane potential of the linear additive neuron (without leak and without refractory period), U_{IF} is the membrane potential of the leaky integrate-and-fire

neuron, U_R is the resting potential of the IF neuron, τ is the membrane time constant of the leaky IF neuron.

When the state of a neuron, or more generally of a dynamic variable, can be analytically computed from any previous state, given no external interference occurred since the previous state, event-driven simulation can be implemented. With this technique, the state of a variable is updated only when its value is required in a calculation or when an external event changes the state of the variable.

For the case of neurons, the state of event-driven neurons is computed only when there is an afferent stimulation that changes the state of the neuron. In such a case, the *t-1* state of the neuron is first computed and then the stimulation is applied during the current time slice (*t*). Although event-driven simulation can greatly reduce the computational effort for large networks, it is only usable for neural models which allow for the analytical integration of the state equations.

In "Neocortex", the event-driven simulation is based on the spike events. Instead of computing each neuron's state, the simulator processes spike lists. It takes all the spikes that were generated during the previous time step and uses those spikes to update only the neurons that are affected by the spikes. However, such spike-driven simulations are only allowed for static synapses and linear additive or leaky integrate-and-fire neurons.

When dealing with large neural systems, researchers may choose to employ different types of models within the same architecture. They might use simple IF neurons for the majority of modules, while complex models only for a few, but critical, neurons. In such cases, iterative and event-driven simulations have to be compatible. Such a compatibility is implemented in the "Necortex" neural simulator.

The "Neocortex" simulation environment uses a global clock with 1 ms resolution. During each time step, all the iterative maps and synapses are updated. At the same time, for event driven maps, the lists of previous spikes are processed, only those neurons being updated that have synapses with the previously active neurons. Compatibility with the iterative maps is assured by calling the update triggers of efferent iterative neurons when a spike-driven neuron fires (Fig. 6).

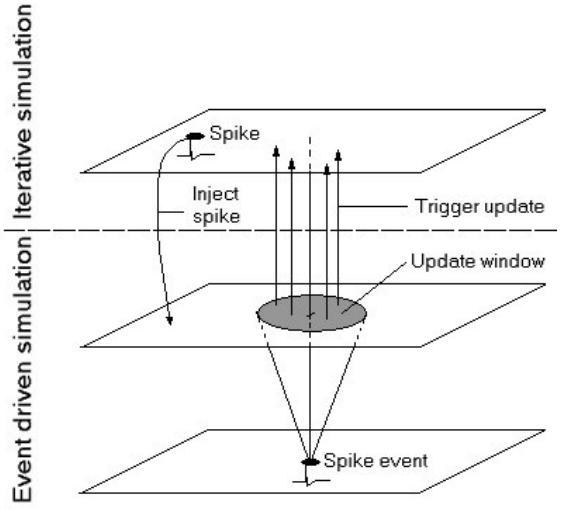

Fig. 6. Interface between event-driven and iterative simulated maps

Also, when a neuron in an iterative map spikes, a spike event is injected in the afferent spike list of the corresponding efferent event-driven map. The global synchronization clock is used both in the iterative simulation and the event-driven updates. The only difference is that in the event-driven case, only the neurons that are affected by the previous spikes are updated during the current iteration.

D. Programming details

From a computational perspective, we tried to maximize the speed of simulation by:

- minimizing the complexity of address calculation during the access to structures in the memory;
- designing data such that caching mechanisms can be effective;
- using only single precision 4 byte floats;
- avoiding virtual functions;
- using massive inline function expansion.

In addition to these techniques, for the event-driven simulation, we used a so-called "accumulation matrix" which stores the PSC computed during the current iteration. First, all the spike events are processed and for each spike-event, instead of updating the state of the target neurons, an effective PSC is accumulated. After all the spikes have been processed, we use the PSC matrix to update the affected neurons (which have PSC $\neq 0$). Such a strategy avoids multiple integrations of the state equation during the same time step, thus accelerating the computation.

Objects communicate with messages that are usually inline expanded functions. Also, we cache addressing invariants before large "for loops" in order to make sure that the compiler doesn't use unnecessary indirections (eg. if we have p->array and we use this address in a large for, we first cache a_cache = p->array and use a_cache[i] in the for loop). We have to mention that such caching is very effective for many older C++ compilers since they do not usually recognize loop invariants.

The processing stream of large arrays of neurons follows the "change the column and then the line" sequence in order to maximize locality and favor caching. The most important variables are forced as "registers" whenever possible, in order to maximize availability for the CPU and minimize memory accesses.

An important observation is that the speed of simulation is greatly improved by such simple programming techniques. When used in conjunction with event-driven simulation and implicit synaptic representation, the speed of simulation can be accelerated thousands of times, making a desktop PC fit for the simulation of millions of neurons with billions of synapses, in real-time [8].

III. RESULTS

We used "Neocortex" for various types of simulations, from simple architectures with only a few neurons, to models with thousands of neurons and millions of synapses.

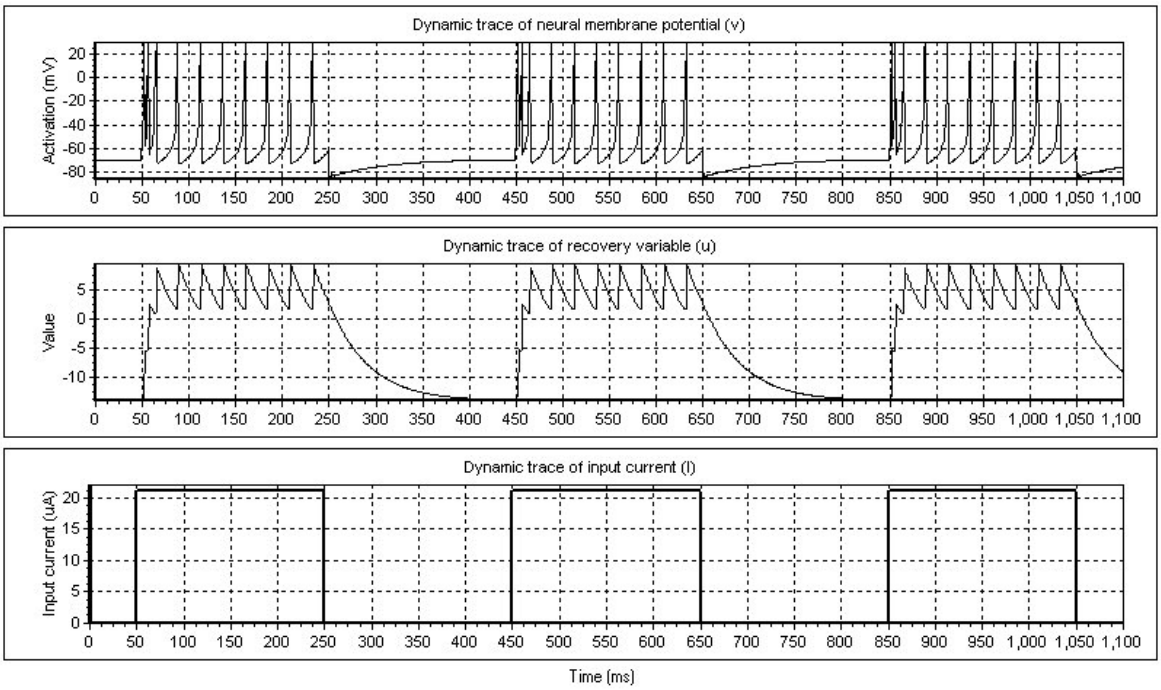

Fig. 7. Stimulation of a regular spiking neuron with square current pulses. The membrane potential (v) and the recovery variable (u) from the model of Izhikevich [5] are plotted as well.

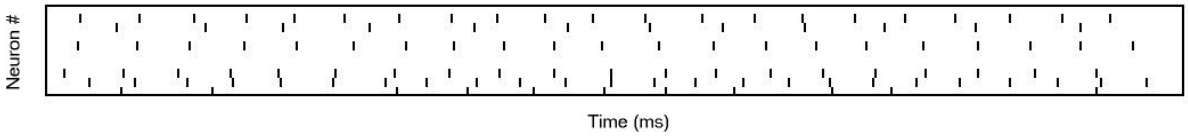

Fig. 8. Time trace of a population of 10 neurons from a neocortical microcircuit.

In general, many neural models lead to simulation problems that are NP-complete, with an exponential explosion of computational effort and required memory (which is mostly the case for the *n to n* connectivity networks). However, if simple models are used, that are fit for event-driven simulation and implicit synaptic modeling, the complexity becomes polynomial or even linear.

We will refer next to the scalability of "Neocortex", that is, the increase of the memory consumption and simulation effort with respect to the number of neurons in the model. Usually, the complexity of the problem of simulation is a function of the number of neurons since the number of synapses is influenced also by the number of neurons.

The required memory for simulation is dominated by the synaptic representation component (since there are usually 10^3 - 10^4 times more synapses than neurons [6]). There is also a neuron memory component. Let M_{syn} be the synaptic component, M_{nrn} be the neural component of the global M_{global} memory consumption (we exclude here the additional components that do not depend on the number of neurons or the insignificant memory consumption).

For the case of implicit synaptic representation, the synaptic memory component is constant or it depends at most in a linear way on the number of neurons:

$$M_{syn\_imp} \approx k; \quad M_{syn\_imp} \approx k * n, \tag{5}$$

The memory overhead is constant when the receptive field sizes are fixed and do not depend on the size of the maps. In such a case, we only need to represent the fixed size connection kernels between maps. However, there are cases when the receptive field size increases with the size of the maps, leading to a linear increase in the size of the receptive fields with respect to the number of neurons. Even in the case of all-to-all connectivity, because of the synaptic sharing, the increase in synaptic memory is linear.

When the synapses are explicitly represented, the memory consumption is much higher. Usually, receptive fields have fixed profiles and fixed sizes so that the memory increase is almost linear (k can be quite high). However, in the all-to-all case, the required memory grows rapidly with the increase of the map sizes (6).

$$M_{syn\_\exp} \approx k * n; \quad M_{syn\_\exp} \approx k * n^2, \tag{6}$$

Usually, both explicit and implicit synapse representations induce a linear increase of memory. The main difference however is the slope k of the linear increase. For the explicit synaptic representation case, k is usually very high (5000 bytes is a normal value).

The representation of neurons requires M_{nrn} bytes of memory. Obviously, M_{nrn} is linearly dependent on the number of neurons (7).

$$M_{nrn} = k_{model} * n, \tag{7}$$

where k_{model} depends on the exact neural model chosen. In "Neocortex", k_{model} is constant due to the separation between the model description and the dynamic variables that describe the state of the neurons (k_{model} = *24 bytes*).

The total memory overhead is:

$$M_{global} = M_{syn} + M_{nrn}, \quad (8)$$

For the case on implicit synaptic representation, the total memory consumption is mainly dominated by the neural component. More, when the receptive field sizes are constant and the number of maps is fixed, the synaptic component is constant too, and small.

However, for explicit synaptic representation, even when the receptive field sizes are kept constant, the memory requirement for representing synapses dominates and quickly grows (Fig. 9). When the receptive field sizes depend on the number of neurons or the number of maps increases, the synaptic memory component quickly explodes.

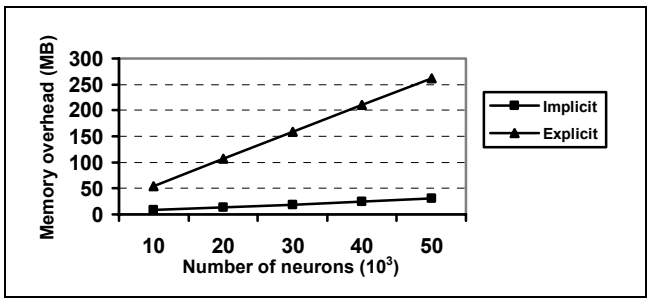

Fig. 9. Total memory consumption for a hierarchic architecture with fixed receptive field sizes. For explicit representation of synapses, the system quickly exhausts the available memory of a desktop computer and also slows down simulation because of frequent cache invalidations.

When event-driven simulation and implicit synaptic representations are used, the simulation is significantly accelerated, real-time simulation of large networks being possible on desktop computers.

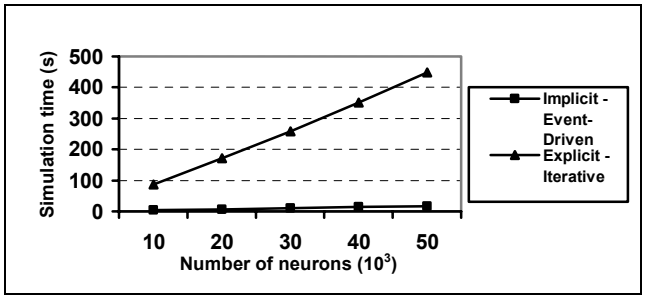

Fig. 10. Dependence of the simulation time on the number of neurons and on the type of simulation / synaptic representation.

We tested the simulator with a 3 layered neural architecture (Fig. 10) and an average receptive field size of 50 units. We used the leaky IF neuron model. First, we wired up the neurons using implicit synaptic representations. Then we simulated the model using the event-driven technique. Because of the small memory consumption and the selective updates, the system is performing very well on a normal desktop computer (few seconds per simulation).

When the same architecture has been generated using explicit synapses, the memory overhead significantly increased. At the same time, we used iterative simulation to test the system's performance in this case. Because of the large memory allocated, caching mechanisms become un-effective and further increase the simulation time. The iterative strategy makes lots of unnecessary updates so that the simulation slows down very much.

As a comparison, for 50000 neurons and 2.5 million synapses, the implicit/event-driven simulation is performed in 17 seconds, while the explicit/iterative simulation is performed in 449 seconds. The simulation results are the same in both cases but the implicit/event-driven simulation is accelerated 26 times in comparison with the explicit/iterative simulation.

We conclude that the best strategy is to use simple models for the large populations of neurons and employ event-driven simulation with implicit synapses whenever possible. At the same time, the critical modules can be modeled with refined detail and interfaced with the larger populations to yield both speed and precision. Flexibility seems to be the most important quality of a neural simulator.

IV. ACKNOWLEDGMENT

The authors wish to thank to Nivis Research (www.nivis.com) for supporting this research.

V. REFERENCES

[1] J.M. Bower, D. Beeman, *The book of GENESIS,* Springer-Verlag , New York, 1998.

[2] P. Dayan, L.F. Abbott, *Theorethical Neuroscience*, MIT Press: Cambridge, MA, 2001.

[3] A. Delorme, et al., "SpikeNET: A simulator for modeling large networks of integrate and fire neurons", In J.M. Bower (Ed.), *Computational neuroscience: Trends in research 1999, Neuro-computing,* Elsevier Science, Amsterdam, Vols. 26–27, 1999, pp. 989–996.

[4] W. Gerstner, W.M. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity*, Cambridge University Press, New York, 2002.

[5] E.M. Izhikevich, "Simple Model of Spiking Neurons", *IEEE Transactions on Neural Networks*, 14, 2003, pp. 1569- 1572.

[6] M. Mattia and Del Giudice P., "Efficient Event-Driven Simulation of Large Networks of Spiking Neurons and Dynamical Synapses", *Neural Computation*, 12, 2000, pp. 2305–2329.

[7] R.C. Mureşan, "Complex Object Recognition Using a Biologically Plausible Neural Model", *in: Advances in Simulation, Systems Theory and Systems Engineering*, WSEAS Press: Athens, 2002, pp. 163-168.

[8] R.C. Mureşan, "RetinotopicNET: An Efficient Simulator for Retinotopic Visual Architectures", in *Proceedings of the European Symposium on Artificial Neural Networks*, Bruges, Belgium, April 23-25, 2003, pp. 247-254.

[9] R.C. Mureşan, I. Ignat, "Principles of Design for Large-Scale Neural Simulators", *Proceedings of AQTR,* May 13-15, 2004, Cluj-Napoca, Romania.

[10] W. Senn, H. Markram, M. Tsodyks, "An algorithm for modifying neurotransmitter release probability based on pre-and postsynaptic spike timing", *Neural Computation*, 13, 2000, pp. 35–67.