A Compiled Accelerator for Biological Cell Signaling Simulations

John F. Keane
Cell Systems Initiative
University of Washington
Seattle, WA USA
jfkeane@u.washington.edu

Christopher Bradley
Computer Science and Engineering
University of Washington
Seattle, WA USA
bradleyc@cs.washington.edu

Carl Ebeling
Computer Science and Engineering
University of Washington
Seattle, WA USA
ebeling@cs.washington.edu

ABSTRACT
The simulation of large systems of biochemical reactions is a key part of research into molecular signaling and information processing in biological cells. However, it can be impractical because many relevant reactions are modeled as stochastic, discrete event processes, and the complexity of the computing task scales with the number of discrete events in a simulation. Traditionally, such simulations are computed on general purpose CPUs, and sometimes in networks of such processors. We show that an alternative algorithm to the conventional approaches based on the Gillespie algorithm reveals a fine-grained parallel structure that is amenable to realization in FPGA hardware. A method is shown for compiling biochemical reaction systems into corresponding Verilog descriptions of simulators that employ this alternative algorithm. We describe a preliminary implementation of such a compiled accelerator that demonstrates the performance of this approach, achieving an initial performance that is 20 times faster than a competing general purpose CPU.

Categories and Subject Descriptors
C.3 [**Special-purpose and Application-Based Systems**],
J.3.a [**Life and Medical Sciences**]: Biology and genetics

General Terms
Algorithms, Design

Keywords
Biology, cell, reactions, reconfigurable hardware, simulation

1. INTRODUCTION
The prevailing framework for quantitatively describing signaling processes in biological cells is the system of reaction equations. A typical reaction equation, as shown in (1), describes a reversible process in which reactant species S_1 and S_2 react at rate k_f to form a product S_3, and S_3 breaks down into S_1 and S_2 at rate k_r.

$$S_1 + S_2 \underset{k_r}{\overset{k_f}{\rightleftharpoons}} S_3 \tag{1}$$

Many simulation platforms[1] have been created for computing simulations of biochemical reaction models. Elementary chemical kinetics modeled by the law of mass action, as in this example, are normally described with ordinary differential equations when the reactants are assumed to be highly concentrated and well mixed. For the example in (1), the forward reaction would be described by (2), where "S_1","S_2" and "S_3" refer to the concentrations of those species.

$$\frac{dS_3}{dt} = k_f \cdot S_1 \cdot S_2 \tag{2}$$

When mixing is insufficient to ignore the spatial structure, partial differential equations are used.

1.1 Modeling with Stochastic Discrete Event Processes
When chemical concentrations are low, the quantization of concentration due to the discrete nature of molecules and their discrete state changes is modeled with whole-number-valued state variables and memoryless stochastic state changes (i.e., reactions). The stochastic discrete event process corresponding to the example in (1) is described by (3),

$$S_1 + S_2 \xrightarrow{k} S_3 \quad = \quad Poisson\left(k \cdot S_1 \cdot S_2\right) \tag{3}$$

where "S_1", "S_2" and "S_3" refer to the count of each species of molecule, and k is scaled accordingly. The Poisson rate parameter, $k \cdot S_1 \cdot S_2$ in this example, is also called the *propensity* of the reaction [6]. Insufficient mixing requires the model to include stochastic diffusion (i.e., random walks).

The complexity of computing simulations of these stochastic discrete event models is approximately proportional to the number of discrete events that are simulated. Thus, while small models (e.g., the reaction in (1), with 100 molecules in the system, where

FPGA'04, February 22–24, 2004, Monterey, California, USA

[1]Examples include Virtual Cell, BioSpice, Gepasi, E-Cell, Mcell, StochSim, Karyote, DBSolve, Jarnac, Dynafit, Ingeneue, JSIM, Stochastica, Cellerator, BioDrive, STOCKS and SigTran

there may be less than 1000 discrete events over the course of a simulation) are trivial to compute, whole cell models are intractable since they are trillions of times larger. For example, the number of discrete events that occur during the 30 minutes between cell divisions for E.coli may exceed 10^{14} [4]. A model like this would take 30 years to simulate on a single CPU (e.g., a 2 GHz Pentium 4). For models that include spatial extent, the computing task is substantially larger, since the diffusion processes dominate the model. Furthermore, multiple simulation runs are needed to acquire statistics from stochastic models, making the computing job even larger.

1.2 Existing Algorithms

Simulating a stochastic discrete event reaction system consists of iterating the following process:

- determine when the next reaction happens
- determine which reaction happens next
- update the quantities of the molecular species according to the reaction.

Several algorithms have been proposed for accomplishing this for Markov processes like the reaction system. In Gillespie's *direct method*, the propensity of each reaction is calculated, and the sum of all propensities is normalized to one. A random number is used to select from the reactions, where the probability of selecting a particular reaction is proportional to its propensity. A second random number is used to determine the time of the next reaction. Gillespie's *first reaction* method draws a random number for each reaction based on its propensity, then chooses the one with the smallest time interval to the next reaction. [6]

Gibson's *next reaction* method uses a similar strategy to the *first reaction* method, computing the time to each of the candidate next-reactions. This method utilizes the property that the remaining random numbers are independent and uniformly distributed after removing the smallest one from each of them and normalizing (i.e., $\tilde{\alpha}_i = (\alpha_i - \alpha_0)/(1-\alpha_i)$, where α_0 is the smallest random number in the current iteration, α_i is the random number associated with the i^{th} reaction for the current iteration, and $\tilde{\alpha}_i$ is the random number associated with the i^{th} reaction for the next iteration). By recycling random numbers from the previous iteration, this method uses only a single new random number for each iteration. Employing a priority queue for the reactions minimizes the other overhead for each iteration. [6]

Gibson's next reaction method reduces the amount of superfluous computation to a minimum and would appear to be a good starting point for a hardware implementation. However, the algorithm is quite complex and requires a dynamically determined set of computations to be performed each simulation step. The complexity of this control along with the heavy use of floating point arithmetic to compute the reaction probabilities has a heavy cost in terms of both compute time per reaction event and amount of hardware. While optimized for execution on a general-purpose processor, it does not make use of the configurable, fine-grained computation available in FPGAs.

2. HARDWARE-SUITABLE ALGORITHM

Our objective was to accelerate simulations by using fine-grained parallelism. By re-examining the problem and being mindful of the costs of implementing different processing elements in programmable hardware, we developed a strategy for compiling reaction system simulator models into programmable logic. Traditional algorithms use hardware intensive floating point multiplication and addition to calculate propensities, transform uniform random numbers into exponential ones, and sort and scale random numbers. The net effect, however, is to choose the next reaction, decrement the whole-number-valued reactant counts, and increment the whole-number-valued product counts. This simpler description of the reaction process hints at how it may be amenable to relatively simple processing in programmable hardware.

2.1 Algorithm

The goal of our algorithm is to reduce the number of clock cycles per reaction event by reducing the complexity of the computation, thus allowing fine-grained parallelism to be applied. This is performed by building separate, simple processors in hardware to handle each reaction, allowing all of the reactions to be simulated simultaneously. The key to implementing a parallel system of processors in FPGAs is to realize the process with *simple* building blocks. This is accomplished by performing stochastic arithmetic on bit streams rather than deterministic arithmetic on larger numbers. Under this strategy, the computing elements consist primarily of random number generators, counters, and simple logic gates. There is no need for traditional arithmetic logic units, and there are no floating point variables.

The approach used in this solution begins with discretizing the reaction processes in time, so that discrete events can only happen at uniformly-spaced discrete instants in time (spaced by Δt). The Poisson process in (3) is closely approximated by a Bernoulli random process, (4), for each sufficiently-small discrete time step, Δt. The probability of an event at any given discrete time step, (5), is still governed by the propensity: the product of the reactant variables and a rate constant.

$$S_1 + S_2 \xrightarrow{k} S_3 \Big|_{\Delta t} = Bernoulli\left(k \cdot S_1 \cdot S_2 \cdot \Delta t\right) \qquad (4)$$

$$0 < P\left[S_1 + S_2 \overset{k}{\rightarrow} S_3 \right]_{\Delta t} = k \cdot S_1 \cdot S_2 \cdot \Delta t \ll 1 \qquad (5)$$

We illustrate our method with a second order (2-reactant) system, however this approach generalizes to higher-order reactions by including additional reactants in the same manner.

Another important part of our approach is eliminating the explicit floating-point multiplication of the factors that comprise the propensities (e.g., $k \cdot S_1 \cdot S_2$), which must occur in the traditional

methods each time an event changes a species count. We also eliminate the need to sum the propensities after each event, a step that is necessary in other approaches. We accomplish this by employing stochastic multiplication, as follows.

A Bernoulli process with probability p can be produced directly from a Bernoulli distributed random variable, or it can be generated from the combination (logical "AND") of a set of independent Bernoulli processes with rates that multiply to form p (e.g., $p_0 \cdot p_1 \cdot p_2 = p$). Therefore,

$$P\left[S_1 + S_2 \xrightarrow{k} S_3 \right]\Big|_{\Delta t} \qquad (6)$$

is equivalent to some $P_0[\cdot] \bullet P_1[\cdot] \bullet P_2[\cdot]$, where $P_1[\cdot]$ is proportional to S_1, $P_2[\cdot]$ is proportional to S_2, and $P_0[\cdot]$ is proportional to k. This allows us to rewrite (5) as (7), where X_i are discrete uniform random variables on $\{0,1,\ldots,M_i\}$, with $S_i \le M_i$ and $k_0 \le M_0$. $\{k_0, M_0\}$ are chosen after the other M_i are chosen to scale the product of probabilities and satisfy (7).

$$P[X_0 < k_0] \cdot P[X_1 < S_1] \cdot P[X_2 < S_2] = kS_1S_2 \cdot \Delta t \qquad (7)$$

The masks, M_i, are chosen to be powers of 2 so that the required range of random numbers can be trivially generated by masking a much larger, fixed length random number. The masks, M_i, determine upper bounds on the species counts, S_i, and on the (scaled) rate constant, k_0. If any S_i becomes larger than its mask, M_i, during a simulation run, the values of M_i (and possibly k_0) must be modified to ensure that $S_i \le M_i$.

In hardware, this product of three probabilities simplifies to the logical AND of the three Bernoulli random variables, which are generated by comparators that perform the less-than operations (Figure 1). Three independent random number generators are required, one for each species and one for the rate constant k_0; however, all of the multiplications and additions are reduced to 3 comparator operations and one 3-input AND operation. As a result, random numbers are produced from the desired distribution *without explicitly multiplying or adding any parameters*.

Note that a system of mass action reactions may be simulated in parallel, as long as no more than one dependent reaction occurs in each time step. As many independent reactions may be performed in a single time step as exist in the system.

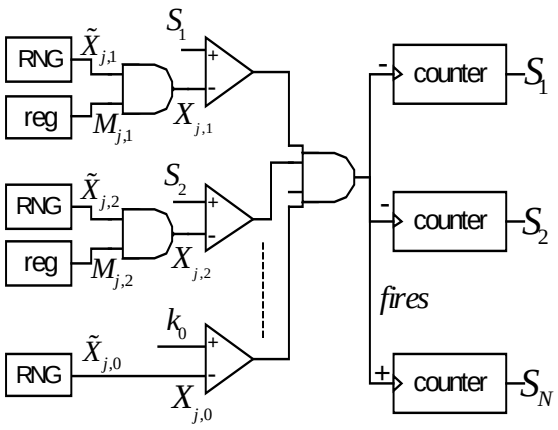

Figure 1. Block diagram of simulation algorithm for the j^{th} reaction in the system. In this reaction, $S_1 + S_2 \xrightarrow{k} S_N$, where additional reactants can be inserted along the dashed line.

The masks, $M_{j,i}$, where j is an index into the set of reactions in the system, are initialized to the next power of 2 larger than the corresponding S_i. If S_i becomes less than $M_{j,i}/2$, the mask can be reduced by a factor of 2, while if S_i becomes larger than $M_{j,i}$, the mask must be increased by a factor of 2. The fastest reaction in the system then determines the value of $M_{j,0}$, corresponding to the rate constant k, according to the allowable approximation error.

2.2 Adaptive Time Step

The Bernoulli approximation converges to the appropriate (exact) Poisson density as the time step becomes arbitrarily small [1]. While a small time step lowers the approximation error, it also increases the simulation time. The choice of time step determines a compromise between fidelity and speed. Since the reaction propensities change according to changes in the species counts in the system, a time step that is optimal for one system state may no longer be optimal when the system state changes. That is, if the reaction rate of the system falls too low, the time step may be increased, while if it increases to the point that conflicting reactions occur during a time step, the time step must be decreased. Therefore, it may be necessary to dynamically update the time step while the simulation executes, to maintain a consistent level of performance.

2.3 Discussion and Previous Work

Previous attempts have been made at accelerating memoryless, stochastic discrete-event systems. In particular, models of basic memoryless queuing processes have been accelerated in FPGAs [2, 3, 9, 10]. These queuing processes are either zeroth order (M/M/1, where the departure rate is independent of how many customers are in the (non-empty) queue, or first order (M/M/infinity, where the departure rate is proportional to the number of customers in the queue). Previous approaches that accommodated zeroth and first order queuing system models are not suitable for extending to higher order mass action processes. First order mass action systems obey the same dynamics as M/M/infinity queuing systems. However, mass action systems are more general and may obey higher order behaviors, as shown in Table 1. In principle, our approach can accommodate any of the systems shown in Table 1, however our present implementation was designed to handle 1st, 2nd, and 3rd order reaction systems, which are the only classes of mass action systems that have ever been observed [7].

Table 1. Examples of Different System Orders. For the queuing systems, μ is the (average) service rate per server, and *N* is the number of customers in the queue. For the reactions, A and B are the number of reactants in the system, *k* is the forward rate constant of the reaction, and *a* and *b* are number of A and B molecules, respectively, needed in the reaction.

System	Rate	Order
M/M/1 (queue)	μ, (N>0)	0
M/M/infinity (queue)	μN	1
A→B (reaction)	kA	1
A+B→C (reaction)	kAB	2
A+B+C→D (reaction)	kABC	3
AA+bB→C (reaction)	kA^{a}B^b	a+b

3. HARDWARE IMPLEMENTATION

Our algorithm is implemented directly in hardware as shown in Figure 2, using 3 clock cycles to execute each iteration (i.e., one time step, Δt). Although the structure is the same for all reaction systems, the number and parameters of each component and their interconnection are different. That is, each reaction system must be compiled into the application-specific hardware that simulates it, as described in the Model Compilation subsection.

The hardware is implemented using three basic components. The counts of each species in the system are maintained in a set of registers called the species counters. In the first clock cycle of each iteration, the reaction event generators decide which reactions happen during the current time step. In the second clock cycle, based on the set of reactions that occur, the event monitors produce a value to add to each species counter and a value to subtract from each species counter. In the third clock cycle, the species counters are updated using these increment/decrement values.

This hardware relies on time step and rate constant dependent parameters calculated by the host machine. Whenever the time step needs to be changed, the hardware must halt and request new parameters from the host.

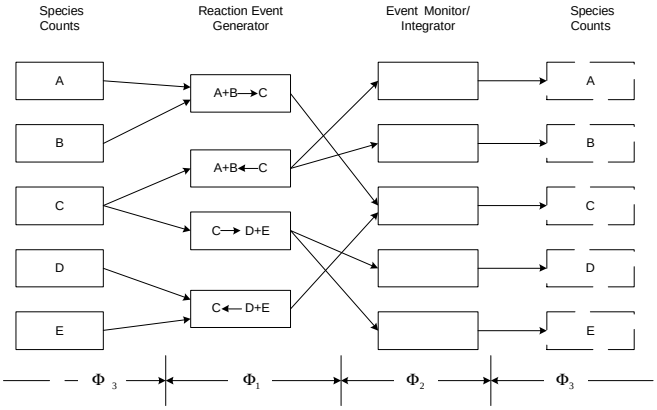

Figure 2. Block diagram of the simulation algorithm, shown for the system $A+B \underset{k_r}{\overset{k_f}{\rightleftharpoons}} C$, $C \underset{k_r}{\overset{k_f}{\rightleftharpoons}} D+E$.

3.1.1 Reaction Event Generator

Each forward and each reverse reaction in the model has a corresponding reaction module which determines whether that reaction occurs during a given time step. The reaction module is parameterized to handle reactions of up to 6 input reactants.

For the reaction $S_1 + S_2 + \cdots S_N \overset{k}{\rightarrow} S_{N+1}$, the reaction module generates a Boolean output called "*fires*" that indicates whether the reaction happens according to (8). In this equation, each $\tilde{X}_i$ is a 32-bit pseudorandom number, each S_i is the species count of the i^{th} input to the reaction, each M_i is a binary mask of the form $2^n - 1$, and k_0 is chosen based on the rate constant and the mask values.

$$fires = \left(\tilde{X}_0 < k_0\right) \&\& \left(\left(\tilde{X}_1 \,\&\, M_1\right) < S_1\right) \&\& \left(\left(\tilde{X}_2 \,\&\, M_2\right) < S_2\right) \&\& \cdots \&\& \left(\left(\tilde{X}_N \,\&\, M_N\right) < S_N\right) \quad (8)$$

Assuming that all counts are less than their corresponding masks (i.e., $S_i \le M_i$ for all species), the probability of the reaction module asserting its *fires* output is

$$P[fires] = \left(k_0 / 2^{32}\right) \cdot \left(S_1 / (M_1 + 1)\right) \cdot \left(S_2 / (M_2 + 1)\right). \quad (9)$$

The host software is responsible for calculating and initializing the values of k_0 and the masks so that for the chosen time step:

$$P[fires] = P[reaction] = reaction\_rate \cdot timestep \cdot count_1 \cdot count_2 \cdot \cdots \cdot count_N \quad (10)$$

The requirement that $S_i \le M_i$ must be checked continuously at runtime. If this is ever violated, the module asserts a "*tooFast*" output to indicate that it can not produce valid outputs with its current settings. This causes the hardware to halt and interrupt the host, asking for a smaller time step.

All the random numbers, $\tilde{X}_i$, are 32 bits wide. They are generated using 167-bit, 2 tap (at 167 and 161) maximal length linear feedback shift registers (LFSRs) which shift out 16 random bits per clock cycle, or 48 bits per iteration, only 32 of which are used. This implementation was chosen because it is both compact and generates good random numbers. By taking advantage of shift register look-up tables, Virtex SRL's, for storage, each random number generator uses a total of just 33 LUTs and 16 registers.

The quality of the random numbers was examined by testing with "DIEHARD" [5] and "ENT" [8], two general-purpose random number test suites. They were also tested indirectly by comparing the output of the reaction modules for convergence with the expected firing frequency. The maximal length LFSR of 167 bits is being shifted by 48 bits per iteration, resulting in a sequence that is still maximal length ($2^{167}-1$), however a minor consequence is that a shifted copy of the same bit pattern repeats every $1/48^{th}$ of the sequence.

3.1.2 Event Handler/Integrator:

For each reaction that occurs, some reactant species are consumed and some product species are produced. For each species, an "event handler" module collects the relevant *fires* signals from the reaction modules that affect it and produces two binary values, one to add to the species counter, and one to subtract from it.

"Collisions" occur when more than one reaction tries to consume the same molecule in the same time step. Collisions are consequences of how this approach discretizes the reactions (in time). When two reactions consume the same species in the same time step, with some non-zero probability they (attempt to) consume the *same copy* of that species of molecule. Depending on the quantity of the species present, this probability ranges from very small, with negligible effect, to very large, with dire consequences. In the worst case, the reactions may consume more of one species than the system contains, leading to negative counter values. When a collision occurs, the simulation pauses and waits for the host to decrease the time-step before continuing. Two rules are implemented to deal with this:

1. If the species counter is less than the number of decrement signals (i.e., the number of times a species appears as a reactant in the system of reactions), collisions are disallowed. This prevents negative species counts.
2. Where a species has a count of N, a collision is allowed if there have been no collisions on that species in the last (approximately) $128/N$ iterations. This allows occasional collisions to happen without calling for a change to the time step. Collisions are permitted more frequently when the species counter is higher because the colliding reactions are less likely to be trying to consume the same instance of the species.

When the module detects a collision that is disallowed, it asserts its "*panic*" output. The panic signals from all the event monitors are OR'ed and used halt execution before the third clock cycle of the iteration, when the species counters are updated, effectively aborting the current time step. The host then reduces the time step, modifies the appropriate mask values and resumes execution.

3.1.3 Species Counters:

Each species count is maintained by a "counter" module. It comprises a register, two small adders for adding in the increment and decrement values from the event monitor, and some I/O and breakpointing logic. The breakpointing logic allows the host to set an upper or lower bound (but not both) for the counter.

3.1.4 Host Interface

The host interface allows the host to initialize a simulation, monitor the simulation state and handle requests for changes to the time step. The interface is based on a simple memory-mapped access to all of the simulation state, which can be viewed while the simulation is running.

The simulation contains registers that track the current time and number of clock cycles. The user can specify breakpoints on these registers as well as each of the species counters, which allows the user interface to specify when the simulation should be halted. Whenever the simulation halts, it raises an interrupt and sets bits in a status register to indicate the cause. The host can also halt the simulation arbitrarily by clearing the run bit in the control register.

The simulation will also halt when exceptional conditions are encountered:

Too Slow: The hardware will request a longer time step if reactions happen too infrequently. The host can set two values to control this, A (average) and M (maximum). At each iteration, a counter is incremented if no reaction happens. If a reaction does happen, A is subtracted from the counter. If the result is negative, it is set to zero. If the counter ever exceeds M, the simulation halts and requests a longer time step.

Too Fast and Panic: The system halts if any reaction module asserts a "*tooFast*" signal, or if any collision handler asserts a "*panic*" signal. Both situations call for a decreased time step.

3.1.5 Model Compilation

Reaction system models must be first compiled into the hardware implementation described in the preceding sections. We have developed a compiler that reads the model description in SBML (Systems Biology Markup Language) and generates a Verilog file comprising the necessary reaction modules, collision modules, and counter modules, with their Verilog parameters, along with

their interconnections. In addition to the model-specific structure, the hardware contains the Host interface, generic I/O and monitoring modules, and the simulation sequencer. We use the Synplicity and Xilinx synthesis and physical design tools to compile this Verilog file into a configuration file for the FPGA.

The model compilation takes a matter of seconds, but the remaining tool chain takes anywhere from a few minutes to over an hour for large models. However, once a model is compiled, it can be used with a wide variety of initial conditions since it depends only on the structure of the model and not on the rate constants or the initial state.

3.2 Implementation

The algorithm is currently implemented using an Annapolis Wildcard II[2], which is a CardBus[3] device containing a Xilinx[4] XC2V3000-4 FPGA along with a large memory which is not currently used.

Host Input/Output (I/O) is carried out with the programmable I/O (PIO) type functions of the Wildcard. The Wildcard also supports direct memory access (DMA), but it was not used due to limited documentation. In the future, DMA can be used to improve performance.

The host is able to read and/or write all of the important registers using a random-access memory-mapped addressing scheme. This is implemented with one bus for addresses and write data, and a tri-stated bus shared by all modules that respond to reads.

The PIO API for the Wildcard does allow blocks of data to be read or written. Reading and writing in the largest blocks possible is currently how we achieve sufficient performance. To take advantage of these block reads and writes, we have implemented methods that allow lists of arbitrary addresses to be accessed as if they were a continuous block..

Registers for species counts are treated specially to allow concurrent simulation and data monitoring. Each species counter has a shadow register that is used when reading back the species counts. A special command causes all of the shadow registers to simultaneously latch the values of their associated counters. In this way, the simulation state can be sampled at frequent intervals and uploaded to the host without stopping the simulation. As long as the time to upload this data is less than the sampling interval, data logging can be done without slowing down the simulation.

[2] Annapolis Micro Systems, Inc., 190 Admiral Cochrane Drive, Suite 130, Annapolis, Maryland 21401 USA

[3] Registered trademark of PCMCIA/JEITA.

[4] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124-3400, USA

3.2.1 User Interface

We have implemented a software interface to our simulator that allows systems biologists using BALSA[5], a GUI developed to allow easy creation and simulation of reaction models, to use our hardware to accelerate their simulations. Our interface provides the following functionality:

- Load the hardware configuration file for a model.
- Set the initial species counts.
- Calculate the initial time step and the related reaction module mask values.
- Start and stop a simulation.
- Set breakpoints based on simulation time or species count values.
- Set sampling intervals for specific species counts. This uses the shadow registers to perform automatic logging of the desired system state.
- Respond to interrupts from the hardware requesting new time step calculation.

In addition to the model's state, the hardware provides access to registers for:

- System status (is the simulation running and what caused the last interrupt)
- The number of iterations executed
- The number of iterations when reactions took place
- The number of clock cycles wasted while the simulation was halted waiting for the host (64 bits).

4. RESULTS & COMPARISON

To demonstrate the performance of our simulator, we used the family of signaling cascade models described in [6] and illustrated in Figure 3.

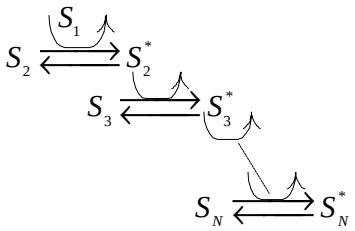

Figure 3. Structure of models used for performance measurements

The measured performance of the hardware simulator is shown in Table 2. These results are measured using a clock rate of 50MHz. The achievable clock rate decreases as the number of reactions in the system is increased: While small reaction models can run at 100 MHz, a full chip with 40 reactions and 60 species can only

[5] Cell Systems Initiative at the University of Washington

run at just over 50 MHz. The additional delay is due to the many-to-many communication between the species counts and the reaction modules, and the reaction modules and the event monitors. The clock rate could be increased by pipelining this communication. We estimate that by adding one more clock cycle to the iteration and retiming, we could increase the clock rate from 50MHz to 80MHz for a performance increase of 20%.

We quantify the performance of our FPGA based simulator using an estimate of the average number of reaction events computed per second. The actual number of events is not captured in the simulator since multiple events can happen per time step, so the event rate is estimated based upon the instantaneous reaction propensities at each logged data point.

The event rate shown in Table 2 represents the "target rate" that would be achieved if the two sources of overhead in this setup (re-computing the time step off-chip and, and communication for data logging) were eliminated.

Table 2. Performance of FPGA-based simulator compared with a software simulator (SigTran) running the Gibson algorithm on a 2 GHz Pentium 4.

	Model				
Species	2	8	16	32	64
Reactions	1	4	8	16	32
Time step changes	7	29	30	26	34
Time steps	3,934	17,079	25,457	32,824	68,480
Simulation time (microseconds)	236	1,025	1,527	1,969	4,109
Steps/change	562	589	849	1,262	2,014
Busy steps (%)	16	17	24	35	36
Events/step	0.16	0.18	0.28	0.45	0.45
Events/s	**2,711,405**	**3,052,482**	**4,695,500**	**7,572,711**	**7,494,402**
Events/s (2 GHz Pentium 4)[6]	333,333	160,000	256,410	285,714	320,000
Performance Gain	**8.1**	**19.1**	**18.3**	**26.5**	**23.4**

The simulator can be in one of three modes: executing the simulation, updating the time step (with cooperation from the host), or performing I/O to log the simulation state. As described previously, the I/O overhead, which currently takes about 70% of the time, can be eliminated using the shadow registers and background uploading of data values via DMA. We anticipate that, in our largest model, 32 model variables could be logged at least once per 100 reaction events, which seems sufficient for most usage cases. Although the hardware supports this, the software interface has not been updated to use it. Ignoring this I/O overhead leads to the performance reported in Table 2. Of course, if the data sampling rate is too high, the data upload time will exceed the simulation time and cause the simulator to wait, reducing the effective simulation rate.

Eliminating the time step recalculation overhead is more difficult. The adaptive time step is presently calculated off-chip by the host computer, which takes a relatively long time due to the communication latency between the host and the WildCard. The number of time step changes that are necessary depends upon the particular model being simulated. Models with large dynamic changes in reaction propensity (especially oscillatory changes) have the greatest need for time step changes. For the cascade models, the average number of reaction events per time step change, over a simulation run, is included in Table 2. This corresponds to a change every few hundred microseconds. Currently, the time step computation and communication takes a few milliseconds. There are several ways to greatly reduce this overhead.

First, the computation, which is fairly simple, can be moved into the FPGA. One option would be to use an embedded processor that would remove the communication overhead. This would reduce the overhead from milliseconds to less than 100 microseconds. This can be reduced further by incorporating the time step calculation into the reaction modules. While this would increase the cost of the reaction modules, it would reduce the overhead to a few cycles. It turns out that most recalculations involve only simple shifts of the mask values and so the increased cost is modest. More complex recalculations that happen infrequently can be sent to the host or embedded processor for handling.

[6] Gibson algorithm as implemented in SigTran

Perhaps the best solution is to cache the mask values for recent time steps on-chip. Generally, the time step is either doubled or halved and thus the same time step may be used many times. By caching values in BlockRAM on the FPGA, the overhead of recalculation can be avoided and the values simply reloaded from memory. Although the values must be transferred sequentially, the overhead can be reduced to a few microseconds. Thus we believe that relatively straightforward improvements to the simulation will achieve the "target rate" performance.

The simulation rate of course depends on the model and the simulation state. The measurements in Table 2 show that for these models a reaction event occurs every 2-3 time steps, a result of setting the time step to keep the collision rate below 1%. Further analysis of how the approximation error depends on the time step may reveal ways to improve this selection. If a rate of one reaction event per time step can be achieved for more complex models on a single FPGA, this would corresponding to a simulation rate of about 17×10^6 events per second. In practice, we have observed peak simulation rates of over 10×10^6 events per second, ignoring overhead.

The space requirements for implementing simulators in FPGAs depend upon the model being simulated. Based upon images that we compiled for biochemical system models, including the ones described in Table 2, the estimated space requirements to implement a model is summarized in Figure 4.

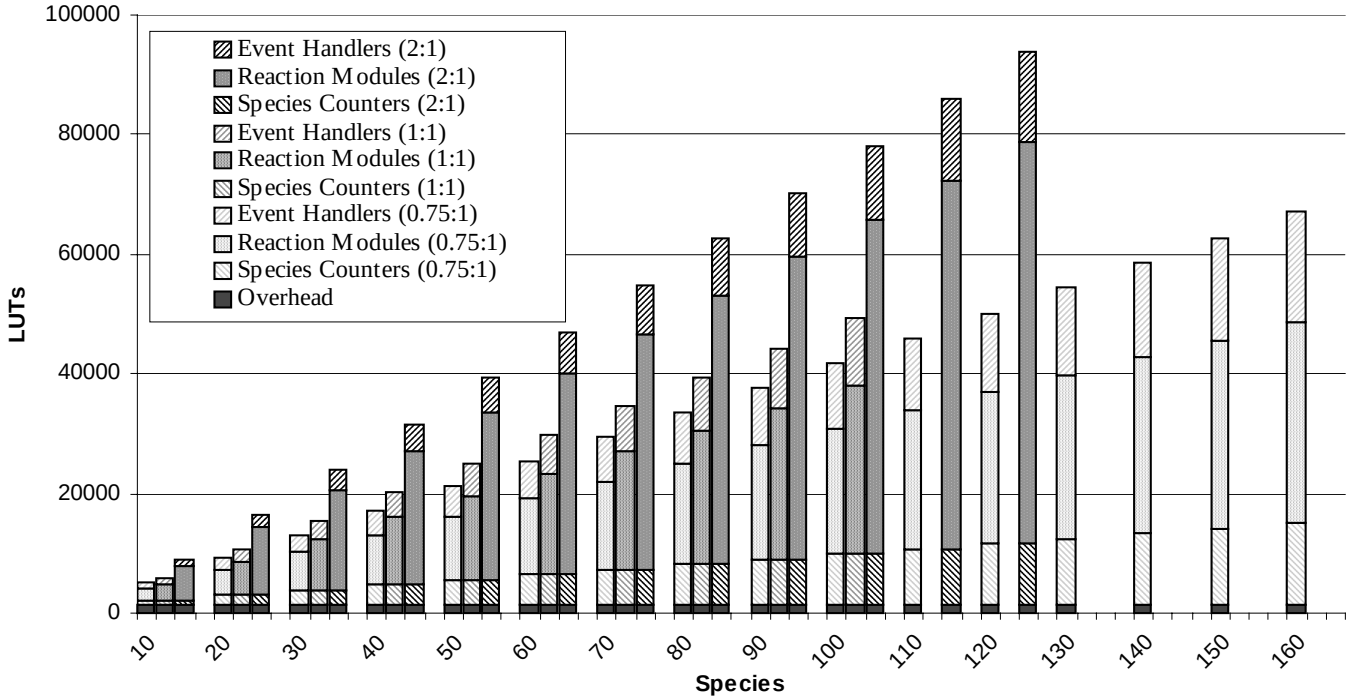

Figure 4. Estimated space requirements for a family of models. Each stacked bar corresponds to one model and is composed of four components: overhead, species counters, reaction modules, and event handlers. For each model size (based on number of different species, along the horizontal axis), up to three variants are shown: reaction-to-species ratios of 0.75:1, 1:1, and 2:1. Note that there are 28,000 LUTs in a Virtex2-3000 , and 90,000 LUTs in a Virtex2-8000.

5. CONCLUSIONS AND FUTURE WORK

By compiling a biochemical simulation into FPGA hardware, we have exceeded an order of magnitude improvement in the computation speed with our initial implementation. Using a single FPGA keeps the cost and form factor within reach of an individual user, empowering the modeler to hypothesize models with much greater interactivity than with traditional approaches.

This accelerator can be useful throughout the modeling cycle, which typically progresses from highly interactive to batch-oriented over the course of a modeling project. It is most suitable for interactive simulation of individual runs (e.g., exploring a model's parameter space), however this single-FPGA solution may also replace clusters of less than 20 PCs for batch operations. For batch operations, its incremental hardware costs are far less than for an equivalent PC cluster, and it requires much less space and power. The number of usage cases that would benefit from the accelerator will increase as we find ways to reduce the compilation time and build libraries of models.

Simulations of models of the size that fit on a single FPGA may run on the order of minutes on a PC, versus seconds using the accelerator. However, when the same model is run 100 times (to acquire statistics, since the model is stochastic) and each of those

ensembles is run over 100 sets of parameters (e.g., searching for optimal values, fitting data, etc.), the whole job may take days or weeks on a PC. This accelerator reduces that to hours. We are currently working on further optimizing the implementation to adapt the time step more quickly and effectively, to increase the clock rate, and to optimize the interaction with the host. We estimate an additional factor of 2 to 4 improvement in performance over the current 20x speedup we are already observing.

Aside from these optimizations of the basic implementation, the next step will be to scale-up our solution so that it can handle the enormous spatial models that are being proposed, which are more appropriate for describing actual cell signaling processes. Although we have pursued an alternative algorithm as described in this paper, we are also investigating how best to implement Gibson's method in hardware. That method has too high a cost for small to moderately-sized reaction systems, however it does scale better to large systems than the method described here and so it is likely that some combination of the two methods will provide the best overall solution.

6. ACKNOWLEDGEMENTS

The authors would like to thank Prof. B. Robert Franza of the University of Washington Cell Systems Initiative and Prof. Les Atlas of the UW Department of Electrical Engineering for their support for this collaboration. We are also grateful to the numerous other participants in this project: Nolan Clark, Cory Crawford, Eugene Lam, Paul Loriaux, Angus MacDuffie, Dr. Michel Pettigrew, Dr. Anamika Sarkar, Avram Wahba, and Kevin West. This work was supported in part by the UW/CSI, PNNL Joint Institute on Cell Signaling, the National Science Foundation REU program, and by research grants and gifts to the UW Cell Systems Initiative from numerous private individuals, from the Washington Research Foundation and from the G. Harold & Leila Y. Mathers Charitable Foundation.

7. REFERENCES

[1] Bertsekas, D. P. and Tsitsiklis, J. N., *Introduction to probability*, Belmont, Mass., Athena Scientific, 2002.

[2] Bumble, M., *A Parallel Architecture for Non-Deterministic Discrete Event Simulation*, Pennsylvania State University, 2001.

[3] Bumble, M. and Coraor, L., "Implementing parallelism in random discrete event-driven simulation", in *Lecture Notes in Computer Science 1388, Parallel and Distributed Processing*, Springer, 1998, 418-427.

[4] Endy, D. and Brent, R., "Modelling cellular behaviour", *Nature*, vol. 409 Suppl., Jan 18, 2001, 391-5.

[5] Marsaglia, G., "*DIEHARD: A Battery of Tests of Randomness*", 1996.

[6] Schwehm, M., Brinkschulte, U., Grosspietsch, K. E., Hochberger, C., and Mayr, E. W., "Parallel Stochastic Simulation of Whole-Cell Models", in *Proceedings of International Conference on Architecture of Computing Systems. ARCS 2002. Trends in Network and Pervasive Computing. Workshop Proceedings*, 2002, 223-31.

[7] Steinfeld, J. I., Francisco, J. S., and Hase, W. L., *Chemical Kinetics and Dynamics*, 2nd ed., Upper Saddle River, N.J., Prentice Hall, 1999.

[8] Walker, J., "*ENT, A Pseudorandom Number Sequence Test Program*", 1998.

[9] Yamamoto, O., Shibata, K., Kurosawa, H., and Amano, H., "A reconfigurable Markov chain simulator for analysis of parallel systems", in *Proceedings of Second Annual IEEE International Conference on Innovative Systems in Silicon*, 1997, 107-116.

[10] Yamamoto, O., Shibata, Y., Kurosawa, H., and Amano, H., "A reconfigurable stochastic model simulator for analysis of parallel systems", in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, 2000*, Napa Valley, CA USA, 2000, 291-292.