# Toward a Fast Stochastic Simulation Processor for Biochemical Reaction Networks

Hyungman Park, Andreas Gerstlauer
Electrical and Computer Engineering, The University of Texas at Austin
{hyungman,gerstl}@utexas.edu

***Abstract*—Computational studies of biological systems have gained widespread attention as a promising alternative to regular experimentation. Within this domain, stochastic simulation algorithms are widely used for in-silico studies of biochemical reaction networks, such as gene regulatory networks. However, inherent computational complexities limit wide-spread adoption and make traditional software solutions on general-purpose computers prohibitively slow. In this paper, we present a specialized stochastic simulation processor that exploits fine- and coarse-grain parallelism in Gillepie's first reaction method to achieve high performance. The processor is designed to support large-scale networks more than a million species and reactions using external DRAMs. In addition, we introduce a dedicated compiler that creates data locality for efficient memory access and data reuse. Our performance evaluation using cycle-accurate simulation shows that our approach achieves orders of magnitude higher throughput for networks with different characteristics of coupling, compared to best-in-class software algorithms on a state-of-the-art workstation.**

## I. Introduction

In recent years, computational biology has become an important tool, promising to enable new discoveries. Toward this goal, one develops a model of the biological system in question, which is in turn simulated on a computer to predict its behavior. A popular class of models are for systems where molecular species interact with another by means of chemical reactions, forming biochemical networks of different sizes. Among different computational approaches, stochastic simulation lends itself well to capturing the random nature of biochemical processes owing to spontaneous fluctuations in the molecular interactions [1]. Generally, the interest is thereby in generating trajectories that simulate network dynamics as the time evolution of species populations.

Gillespie's original stochastic simulation algorithms (SSAs) [2], [3] evaluate reactions in a continuous, stepwise fashion to execute the one most likely to occur next. Since the algorithms simulate individual reactions over time, they are accurate but computationally very intensive. They have an algorithmic complexity of $O(EM)$ for a single simulation run, where $E$ is the number of simulated events and $M$ is the number of reactions. In addition, the need for a large number of Monte Carlo simulations over a long biological time of interest makes traditional software solutions on regular computers or supercomputers either prohibitively slow or expensive. This severely limits the scope of possible studies, especially for large and complex biological networks. For example, simulating expression of one gene in one generation of E. coli (with 30 min. simulated real time between cell divisions) can take more than 20 h [4]. Simulation of the whole cell, which encompasses more than $10^{14}$ events [5], requires 30 years even on modern GHz-class workstations [6].

Various approaches have attempted to tackle the complexity problem either through algorithmic enhancement or by realizing the algorithms on high-performance compute platforms. Many optimized variants of SSAs have been developed, most of which improve performance by leveraging dependency information [7], [8] or by introducing efficient data structures and search methods [9], [10], [11]. Their computational costs vary between $O(E \log M)$ and $O(E)$.

Researchers have also harnessed the compute power of different platforms including supercomputers [12], [13], compute clusters [14], [15], and GPUs [16], [17]. All these approaches utilize multiple compute nodes to exploit the potential massive parallelism exhibited in SSAs, both coarse-grain across multiple simulation runs and fine-grain across concurrent evaluations of multiple reactions in each time step [18]. However, supercomputing-based solutions are expensive. Furthermore, in all cases, large memory, bandwidth, or synchronization demands limit achievable performance.

Previous approaches for custom hardware realizations of SSAs on FPGAs [6], [19], [20], [21] have shown promising results. However, the flexibility provided by reconfigurable hardware fabrics limits their size and performance. Furthermore, FPGAs typically require difficult and time-consuming redesign processes for each new problem instance, which involves complex synthesis tools that are not intuitive to the intended users in the natural sciences. While some approaches allow for reconfiguration without the need to resynthesize [21], they are limited to a particular SSA and impose tight restrictions on parameters such as network size.

Our long term goal is to provide life scientists with computational tools that will allow them to study hitherto infeasible, life-size networks. Toward this goal, we envision special-purpose stochastic simulation processors that combine many low-cost cores into a flexible and scalable design with sufficient performance and capabilities to simulate large-scale biochemical networks. In this paper, we present the hardware design of an SSA processor core that can accommodate more than a million reactions and species using external DRAMs. Our processor is supported by a preliminary version of an optimizing compiler that can map network descriptions in standard SBML form [22] onto the SSA execution fabric. Using cycle-accurate simulations, we compare performance with best-in-class software algorithms running on state-of-the-

TABLE I
ELEMENTARY REACTIONS

| Type | Reaction | Propensity $a_m$ |
|---|---|---|
| Source (Src) | $\emptyset \xrightarrow{c}$ Products | $c$ |
| Unimolecular (Uni) | $S \xrightarrow{c}$ Products | $cx$ |
| Bimolecular (Bi1) | $2S \xrightarrow{c}$ Products | $cx(x-1)/2$ |
| Bimolecular (Bi2) | $S_1 + S_2 \xrightarrow{c}$ Products | $cx_1x_2$ |

art workstations. Furthermore, we show how characteristics of simulated networks can affect design tradeoffs in relation to compiler optimizations and performance-limiting factors such as external bandwidth and data reusability. Results show that our current design running at 400 MHz in cost-effective legacy technology can outperform existing software solutions by orders of magnitude in improved throughput.

The paper is organized as follows: In Section II, we first review various SSAs and choose the algorithm that best suits a hardware implementation based on analytical performance models. In Section III and IV, we elaborate on the details of our design and dedicated compiler. In Section V, we evaluate and compare the performance with software simulation. Finally, in Section VI, we conclude with a summary and an outlook on future work.

## II. ALGORITHM ANALYSIS

In the following, we analyze various SSAs in terms of suitability for hardware acceleration [23]. We use throughput as our comparison metric and define it as the number of simulated time steps per unit simulation time. To achieve speedups, we can parallelize stochastic simulations at two different levels: (i) coarse-grain across multiple instances of a Monte Carlo simulation and (ii) fine-grain across concurrent evaluations of multiple reactions in each time step. Therefore, we can naturally envision our hardware model as a coarse-grain array of stochastic processing elements (SPEs) that each contain a number of fine-grain reaction units (RUs).

In stochastic simulations, a biological network is described as a set of chemical equations having $M$ reactions and $N$ species with an initial state of molecular populations $X = [x_1, \ldots, x_N]$. Each reaction $R_m$ in the network occurs at a rate $c_m$, where a combination of reactant species yields product species, i.e., populations are affected according to a state-change vector $\mathcal{V}$. For example, in a reaction DNA $\xrightarrow{c}$ RNA, DNA is transcribed to RNA, consuming one DNA molecule and producing one RNA molecule at a rate of $c$, thus $\mathcal{V}$ = [-1, 1]. The probability that a reaction will fire in the next infinitesimal time interval $dt$ is $a_m dt$, where $a_m$ is the so-called propensity for reaction $R_m$. Complex chemical equations can be decomposed into elementary reactions each having at most two reactants [1]. Corresponding propensity functions are summarized in Table I [9].

### A. Stochastic Simulation Algorithms

SSAs can mainly be classified into exact, approximate, and hybrid methods. Our focus in this paper is confined to exact SSAs only. In such algorithms, all reactions are repeatedly evaluated at every time step by answering the following two questions: What reaction ($R_\mu$) will fire next and when ($\tau$) will the next reaction occur? Once $R_\mu$ and $\tau$ are found, the simulated time is advanced by $\tau$, and the selected reaction is fired by adding the change vector $\mathcal{V}_\mu$ to the state $X$.

Exact SSAs include the direct method (DM) [3] and the first reaction method (FRM) [2]. In both algorithms, all $M$ propensities need to be evaluated in every time step. However, in a regular sequential implementation, a DM is typically more efficient. It randomly generates $\tau = -\ln(r_1)/a_0$ and samples $\mu$ as the index satisfying $\sum_{m=1}^{\mu} a_m > a_0 r_2$, where $r_1$ and $r_2$ are unit uniform random numbers and $a_0 = \sum a_m$ is the sum of propensities over all reactions. By contrast, an FRM first computes the predicted times $\tau_m = -\ln(r_m)/a_m$ followed by a $\tau$ aggregation, which determines $\tau$ and $\mu$ as $\operatorname{argmin}_m\{\tau_m\}$. This requires a larger number of random number generations, which are expensive in software. In hardware, the situation is reversed. There, computation of $-\ln(r_m)$ and $a_m$ can be performed in parallel and the overhead for generating random numbers is effectively amortized. Furthermore, while the latency for parallel computation of $\tau$ is the same in a DM and FRM, a DM requires an extra search to determine $\mu$. Even when using a parallel comparator with $b$ banks of memory, the cycle count required for this search still grows linearly with $M/b$, significantly increasing the latency on the critical path.

Many optimized variants of the DM and FRM have been proposed. One of the key insights is to only recompute reactions for which propensities are affected by the firing of the previous reaction. This includes the optimized direct method (ODM) [8], the sorting direct method (SDM) [24], and the logarithmic direct method (LDM) [25] as variants of the basic DM, and the next reaction method (NRM) [7] as an optimization of the FRM. Differences exist in the data structures used to maintain an (ordered) history of previously computed reaction data. Such methods can be implemented as derivatives of the basic FRM and DM microarchitectures. Dependency information can be stored as part of the global reaction tables, and after receiving the final state-change vector $\mathcal{V}$ from memory, a central controller can trigger execution only for those reactions for which input species populations have changed. The difficulty lies in realizing the history caches, where we only consider the ODM and NRM as candidates for hardware implementation.

In the ODM case, an additional memory stores the last computed $a_m$ and $a_0$ values. In each iteration, only the propensities $a_i$ that have changed will be recomputed and the sum $a_0$ is updated before following a regular DM computation. In the NRM case, the $\tau$-aggregator is modified to operate as a priority queue that always delivers the index and value of the currently smallest $\tau$. In the original NRM, dependent $\tau_m$ are recomputed by scaling their previous value with the ratio of old and new propensities. This avoids the need for generating a new random number but requires an additional $a_m$ memory with associated control overhead. In hardware, random number generations are effectively hidden and it will be more efficient to simply regenerate new $\tau_m$. This has already been proven to be statistically equivalent [7].

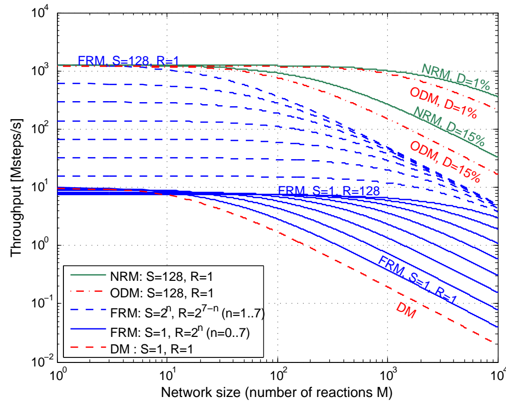



Fig. 1. Throughput comparison of SSA hardware variants.

In dependency-based methods, a high overhead can exist for maintaining the necessary data structures. Furthermore, they are only effective for networks with low dependency ratios. In recent years, several SSAs have been proposed with the goal of achieving near-constant computational complexity in each time step. DM variants using a composition-rejection method (SSA-CR) [11] and partial-propensity-based direct methods (PDMs) [10] group and refactor computations by propensities or species to bound the search space or eliminate the need to recompute unaffected subexpressions, respectively. Both approaches suffer from exponential memory requirements, however, which negates their gains in computational complexity. Finally, a hashing technique for the NRM [9] can achieve constant-time reaction insertions and lookups at the expense of limited history space. Complexity remains a function of the network dependency, however, and the limited size of hash tables requires occasional refills via fallback to a full FRM.

### *B. Performance Analysis*

We have developed analytical performance models for the SSA variants considered for hardware implementation. We collected cycle times of all the needed operators by synthesizing Synopsys Designware libraries using a 45nm technology at the worst corner case. Based on these results, we relaxed obtained delay values by a factor of more than two to target a clock frequency of 400 MHz. We assume that RUs are pipelined to process one reaction per cycle. Depending on the method being realized, the basic $\tau$-processing pipeline in each RU has a latency $L_d$ of 40 to 60 cycles. Furthermore, issue of reaction data and the $\tau$-aggregator are realized by pipelined binary trees that each take $\lg R$ cycles. Therefore, assuming an average network dependency factor of $D$ for the NRM (1 for FRM), computation of $M$ reactions in each time step requires $L_d + 2\lceil \lg R \rceil + \lceil \frac{DM}{R} \rceil$ cycles. DM and ODM computations require additional cycles to sample the reaction index.

Fig. 1 compares throughput of various designs for different configurations in terms of the number of independent SPEs $S$ and the number of RUs $R$ per SPE. For small network sizes, throughput ranges from around 10 Msteps/s for a single SPE up to 1280 Msteps/s for a chip with 128 SPEs. For larger network sizes, single SPE performance approaches a projected peak rate of $R$ reactions every cycle for a maximum throughput of $400R$ million reactions or $400\frac{R}{M}$ million steps per second. We can note that peak throughput for 128 SPEs with a single RU is equivalent to the performance of a single SPE with $R = 128$ (i.e., $51.2/M$ billion steps per second).

As expected, in hardware, an FRM or NRM outperforms a DM or ODM, respectively. In addition, NRMs outperform FRMs for networks with low dependency factors. However, NRM performance degrades for strongly-coupled networks. Moreover, complex data structures can be hard to implement, and limits in on-chip memory will result in either long access latencies or regular FRM fallbacks. For these reasons, we focus on the choice of FRM for initial implementation of our SSA processor, which we will later extend with NRM (with fallback) capabilities. Note, however, that our analysis thus far assumes infinite on-chip memory bandwidth and size. In reality, sustaining species and reaction traffic to keep reaction units occupied is a major concern in SSA computations. A key aspect in our design is therefore the optimization of memory interfaces coupled with caching, prefetching, and compiler support to exploit available locality.

## III. Stochastic Simulation Processor

Fig. 2 depicts a hierarchical view of our stochastic simulation processor (SSP). The processor is organized as a scalable array of stochastic processing elements (SPEs) communicating with external interfaces including a memory subsystem and an output interface. Driven by reaction and species instructions, each SPE containing multiple reaction units (RUs) follows an FRM algorithm to process species populations, determine reactions to fire, and update trajectories in on/off-chip memories. As will be detailed later, RUs include local reaction and species memories, but to enable simulation of large-scale networks, reaction instructions and species data can optionally be stored in external memory. The design employs a memory subsystem to interface with external, off-chip DRAM channels. In addition, trajectories are sent to an external host through the output interface.

With this architecture, we can exploit both simulation- and reaction-level parallelism across and within SPEs, respectively. Networks to be simulated are compiled onto the SSP by partitioning reactions across RUs in each SPE (see Section IV). For each reaction, an instruction is generated and stored in either RU-internal or external reaction memory. Population values for species associated with each reaction are equally stored in either RU-internal or external memory. To avoid access conflicts, the population for a species shared among reactions mapped to different RUs is replicated in distributed, RU-specific internal and external species memories. Coherency is maintained by broadcasting species updates to all RUs.

A key challenge is how to efficiently utilize the external bandwidth shared among SPEs and how to effectively hide associated latencies. The FRM exhibits little temporal locality within the stream of reactions that is processed in every time step. However, access patterns are predictable and, with the help of reordering and renaming in the compiler, both reactions

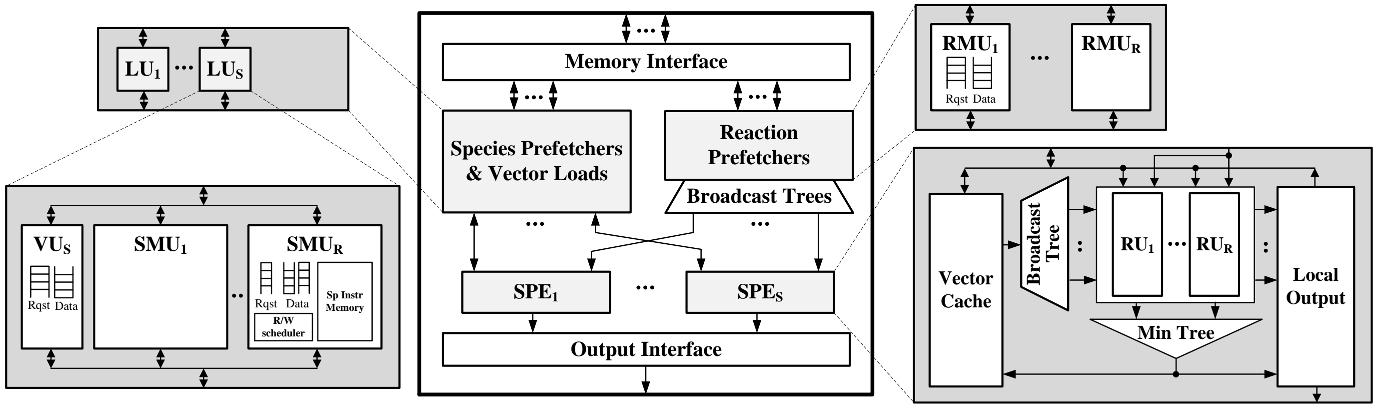



Fig. 2. Stochastic simulation processor (SSP).

and species have strong spatial locality. We have designed the processor with this philosophy in mind. We will explain various design aspects from the perspective of maximizing locality, reusability, and hence exploitation of parallelism.

*A. Memory Subsystem*

The memory subsystem consists of three main parts: a memory interface, species prefetchers, and reaction prefetchers. The memory interface arbitrates read and write requests from clients using a round-robin scheme to access a set of shared, external DRAM channels. Address generation is done in the prefetcher units. The main role of the memory subsystem is to transfer necessary data from and to SPEs. This includes reactions, species indicies/populations, and state-change vectors. A key insight is that even though reactions and species are tightly coupled, we can possibly think of these as two separate entities in the memory subsystem. (How these are split and merged will be discussed later.) Therefore, we can let the memory subsystem independently read and write interleaved streams of different data in order to eliminate dependencies and thus reduce latency.

One interesting feature of the FRM is that the same sequence of reactions is iteratively and repeatedly evaluated. Furthermore, reactions can be fetched in any order. This implies that we can achieve consecutive, non-blocking reads for reactions stored in external memory. In addition, since each SPE simulates independent trajectories of the same network, the read traffic of reactions can be shared among different SPEs, which greatly reduces external memory bandwidth. Reaction prefetchers contain as many reaction memory units (RMUs) as there are RUs in each SPE, labeled with lane IDs from 1 to $R$ in Fig. 2. Each RMU is responsible for continuously feeding read requests into buffers that are read and served by the memory interface. To distribute incoming, shared reaction traffic, broadcast trees configured as $R$-way binary subtrees feed data into SPEs with a latency of $\lg S$ cycles, where $S$ is the number of SPEs. When the prefetched reaction data arrives in an RMU, its associated subtree reads the data from the RMU buffer and broadcasts it to all $S$ RUs with the given lane ID.

Species prefetchers require as many local memory units (LU) as there are SPEs because separate copies of species populations must be maintained for generating each trajectory instance. An implication is that external memory bandwidth is affected both by the total number of RUs ($S \times R$) and the characteristics of the network being simulated. If a network exhibits a high ratio of species to reactions $N/M$, it will increase the species traffic proportionally and thus impacts the overall throughput of the design. Hence, network features can play a crucial role in the number of parallel simulations that can be supported with maximum performance.

In each LU, there are $R$ species memory units (SMUs), each with a memory that stores species instructions, a read/write scheduler, and request/data buffers. Dedicated species instructions, which are processed by SMUs to drive the loading of population data from external memory according to a predefined prefetching schedule, are automatically generated by our compiler. In doing so, the compiler analyzes the input network and renames each species to create strong locality in the species indices. In this way, we can organize prefetching instructions as a concatenation of a species index (*spid*) with the number of consecutive species items (*length*) to fetch. For networks intrinsically having many reactions sharing the same reactants, and as supported by the compiler, this setup can dramatically reduce the number of required prefetching instructions and hence external memory references. In addition to SMUs, each LU includes a single vector memory unit (VU) associated with each SPE. VUs respond to cache misses and corresponding refill requests for the loading of state-change vectors in SPEs (see Section III-B).

The scheduler in an SMU orchestrates the reads and writes of species populations when its client RU issues an update (memory writeback) request. Naively interrupting the continuous stream of memory reads in order to serve writebacks as they are requested is likely to incur additional latency in the DRAM system, e.g., if the addresses exhibit no locality. To fully utilize the open-page policy of the DRAM controller, which maximizes the hit rate of a row buffer in DRAM access, the scheduler is therefore designed to achieve split transactions for the writes, i.e., they are deferred until the next SSA time step and finally scheduled with another read sharing the same DRAM row address. This does not break the correctness of the read/write order because a write is made to occur always

before a read. The frequency of discontinuity in the reads is very rare. Especially in large networks, the firing of a reaction in each time step requires only a few species updates.

### B. Stochastic Processing Elements

A block diagram of the SPE is shown on the right of Fig. 2. The three main functionalities of the SPE are: (i) generation of a ($\tau$,$\mu$) pair for each time step, (ii) loading and broadcasting of a state-change vector $\mathcal{V}$ via the vector cache, and (iii) aggregating species updates from each RU.

RUs take reaction instructions and species data, located in either local on-chip or external off-chip memory, as input and generate streams of $(m, \tau_m)$ values that are in turn passed into a binary tree of comparators (MinTree). When the MinTree is done processing reactions to determine the one with the minimum $\tau$, it notifies the vector cache to load the state-change vector for the reaction $\mu$ to be executed. Depending on whether a cache hit or miss occurs, the vector cache either broadcasts the state-change vector into the RUs, or it first issues a read request to the associated VU in the memory subsystem. The vector cache is included to exploit temporal locality among repeated firings of the same reaction. Such locality exists in many classes of typical networks, as exemplified by ODM [8] and SDM [24] SSAs, both of which leverage such network characteristics to improve performance. Once broadcast into the RUs, state-change vectors are stored in locally and used to update corresponding species values as they are processed in the next iteration. As part of such updates, RUs also issue writebacks to RU-local or external species memories.

SPEs' local output units aggregate species populations that have been updated by each RU. At the end of each iteration, they send aggregated data to the external host via the output interface. In addition, if data is tagged as externally located species, a writeback request is sent to the corresponding SMU of the memory subsystem. The amount of data to aggregate is minimal because every reaction firing only entails an update of a few, typically at most four to five species [7]. Moreover, depending on what reactions are allocated locally to a given RU, external updates may not be needed at all.

The main problem of implementing large-scale FRM algorithms is that there are several factors that can lead to stalls in the RU pipeline. Stalls are created mostly by data hazards in which the current SSA iteration is blocked on species updates resulting from the reaction fired in the previous iteration. This requires a load of a state-change vector followed by an update of species populations and a writeback of all copies of updated species data. Moreover, given a reaction, the RU needs to first figure out what input species are involved in the reaction and then fetch the species populations accordingly. Incurred latencies are especially prominent if the data is placed externally and if a miss occurs on a reference to the vector cache. In our design, we focus on resolving resulting underutilization of the pipelines. Toward this goal, we aim to maximize the data reusability through both hardware and compiler support, including modifications to the FRM algorithm itself.

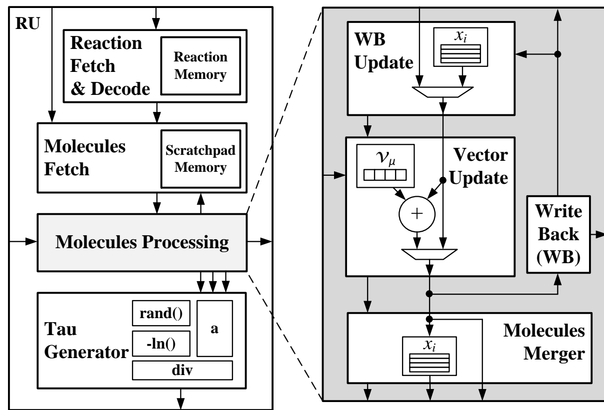



Fig. 3. Reaction unit (RU).

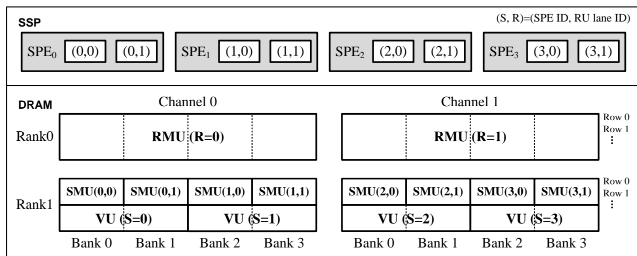
Fig. 3 shows a block diagram of the main RU pipeline. Reactions are fetched in a consecutive manner first from local reaction memory and subsequently from the RMU-internal reaction buffers filled by the memory subsystem. In parallel with reactions, species data is loaded from an internal scratchpad or from the SMU's buffer by a molecules fetch unit. The partitioning of reactions and species as well as the schedule of their prefetching sequences is managed by the compiler. In this way, we can use a scratchpad memory with prefetching instead of expensive caching. Finally, species are processed together with their reactions when both are available.

Forwarding paths in an additional molecules processing unit are used to inject updates into species as they are read. The state-change vector received in the previous iteration is used to update matching species values streamed through a vector update unit. In addition, updated values are written back to local scratchpad or external species memories. Writebacks are sent to external memory via the SPE's local output unit. In addition, they are cached in a local writeback update unit that can substitute externally fetched data until updates, which, as described above, are delayed by one iteration to allow for optimized DRAM accesses, have ultimately been scheduled and committed by the corresponding SMU. Hence, external memory roundtrips are bypassed and continuous prefetching of species across iterations becomes possible, i.e., the processing of the next, already prefetched iteration can begin as soon as the latest change vector becomes available. Finally, prior to passing the species data into the tau generator, if the reaction requires two different reactants, the merger unit associatively searches its internal registers by reaction tag and either passes both reactants to the tau generator or stores a reactant species temporarily until its companion is available.

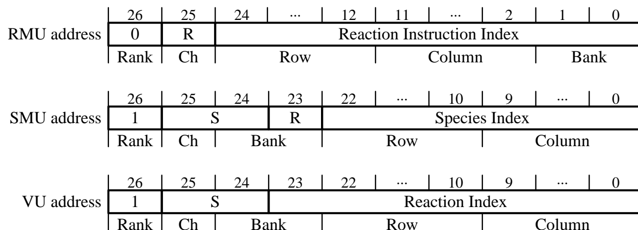
The tau generator then processes species data and reaction coefficients to compute $\tau_m = -\ln(rand())/a_m$ in a fully parallelized and pipelined fashion, where $a_m$ is produced using a dedicated pipeline that implements the equations in Table I.

### C. Mapping of Instructions and Data

Partitioning and allocating external memory space for reactions and species is crucial for achieving performance. Ex-




(a) SSP system and DRAM layout.




(b) 27-bit DRAM address format.

Fig. 4. An example of an SSP configuration and mapping of instructions and data.

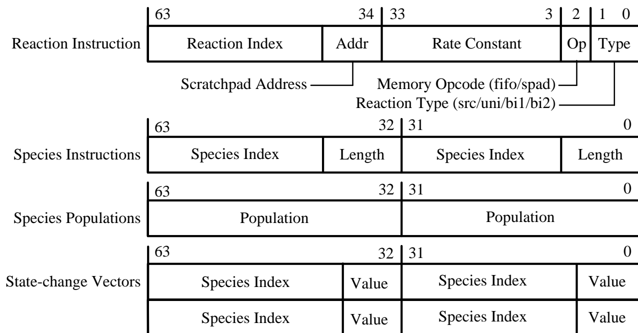



Fig. 5. Format of 64-bit instructions and data.

cessive DRAM bank and page conflicts can create significant latency overhead, thus underutilizing the limited external bandwidth shared by multiple SPEs. We optimize DRAM access patterns through a combination of hardware and compiler support. Given information on the target DRAM system and the SSP hardware configuration (i.e., number of SPEs and RUs), the compiler generates bit files that can be used to configure the SSP. The bit files contain DRAM address bitmasks as well as a segment-based memory maps for all types of data including reactions, species indices/populations, and state-change vectors. The hardware merely generates addresses based on this information using incremental counters.

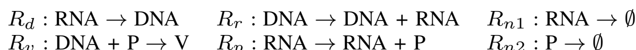
Fig. 5 shows the instructions and data format of the SSP. Each reaction instruction contains information for reaction type, rate constant, and one participating input reactant. A Bi2 reaction requires two such instructions to store additional reactant information. References to reactant data stored in species memories are encoded as *Op* and *Addr* bit fields. If the *Op* is set to *FIFO*, the species data is first read from the SMU's prefetch buffer and cached into the scratchpad location given by the *Addr* field. (Currently, half the scratchpad is used as memory dedicated to internally-mapped species, and the other half is used to cache externally fetched data.) In either case, species data is read from the given scratchpad address and processed by the RU. Species data in the scratchpad is further tagged with its species index to allow for later writebacks.

Other species prefetching instructions, species data, and state-change vectors are simply encoded as relevant information in the instruction or data words, where two species instructions and two sets of species data can fit into each 64-bit word. Note that expanding the bit width of instruction and data words will allow to increase supported network sizes but will only work given enough external bandwidth. Also, the number of reactions and species that can be supported is limited by the size of the target scratchpad memory and the bit width of the reaction instruction. Currently, by constraining the *Addr* field to 10 bits, we can support more than a million reactions. In terms of a species count, more than a million species are also easily achievable.

Finally, Fig. 4 shows how data is partitioned and allocated to the DRAM system. For simplicity, we assume that all species prefetching instructions are mapped into local memory and thus omitted in the illustration. As shown in Fig. 4(a), the compiler distributes different data types across separate channels, ranks, and banks to avoid conflicts among the prefetching blocks of the memory subsystem. In this example, SMUs and VUs share the same rank. This has little impact on performance since a vector load only occurs once per SPE and time step. In configurations with more than two ranks, the compiler will allocate other unoccupied ranks for VUs. Fig. 4(b) shows the resulting address encoding used in the prefetching units. Species and reaction indices stored in the corresponding instructions are thereby mapped into DRAM addresses to read species data and change vectors from, respectively. By contrast, reaction instructions themselves are simply read from consecutive reaction addresses generated in RMUs. As mentioned above, species instructions are only stored in local memories, i.e., never fetched from external DRAM. Since an RMU occupies an entire rank of a channel, the address is mapped to efficiently utilize the bank-level parallelism. Note that since the memory interface currently implements a round-robin arbitration and SMUs take their turns one at a time, their bank access patterns are similar.

## IV. NETWORK COMPILER

Configuring, partitioning, and scheduling of executions on the SSP can have a significant, non-intuitive influence on performance. We have developed a compiler that uses simple heuristics to automatically map network descriptions in standard SBML [22], [26] format onto the SSP. The compiler aims to optimize both the scheduling of reactions and species as well as the partitioning of networks across multiple RUs in an SPE. A key for both is to expose locality through reordering and renaming. Consider a simple model of intracellular viral infection [1] with six reactions and four species:



**Algorithm 1** Scheduling of reactions for off-chip species.

```
1:  while not all_reactions_scheduled() do
2:    for all ru do
3:      for all scratchpad_line_of_data do
4:        line ← bring_in_next_line(ru)
5:        update_scratchpad(ru, line)
6:        op ← FIFO
7:        for type ∈ {uni, bi} do
8:          s ← True // s: schedulable, m: reaction index
9:          while s do
10:           (s, m, addr, spid) = search_reaction(ru, type)
11:           if s then
12:             schedule(m, op, addr, spid)
13:             op ← SPAD
```

Renaming species to ($S_0 \equiv$ RNA, $S_1 \equiv$ DNA, $S_2 \equiv$ P, $S_3 \equiv$ V) and reordering reactions to ($R_0 \equiv R_d$, $R_1 \equiv R_p$ $R_2 \equiv R_{n1}$, $R_3 \equiv R_r$, $R_4 \equiv R_v$, $R_5 \equiv R_{n2}$, $R_6 \equiv R_{dummy}$) creates locality in both the reaction and species indices when executing the network in the given order. Note that even though V is not a reactant species, it has to be streamed through an RU using a dummy instruction $R_6$ (encoded as special reaction type) for cases when updates of V need to be recorded. Spatial locality among monotonically increasing indices helps to exploit parallelism when accessing data in external DRAMs. Furthermore, temporal reuse of species utilizes the scratchpad memory efficiently. In general, however, the presence of bimolecular reactions such as $R_v$ will create more complex dependencies that make optimal scheduling of reactions a much harder problem.

Our compiler executes the following steps: (i) read an SBML model, (ii) partition reactions, (iii) rename and schedule species, (iv) schedule reactions related to on-chip species, (v) schedule reactions related to off-chip species, (vi) create a memory map for external DRAM channels, (vii) create instructions for both reactions and species, and (viii) generate programming files. After reading an SBML model, the compiler distributes reactions across RUs by grouping them based on reactant species. For bimolecular reactions, it arbitrarily chooses the lower-ordered reactant as the grouping factor. To balance workloads, the compiler then partitions groups uniformly across RUs. Next, renaming is done by assigning an index to each species such that those shared by more reactions than others are given lower indicies. This enables the compiler to map these species and their associated reactions to internal memories first, thus extending the period between the time external data is prefetched and needed. To map as many reactions as possible, reactions are scheduled into internal memories in priority of reaction types, with unimolecular reactions before bimolecular ones. Bi2 reactions are only scheduled if both reactants are mapped internally. Other Bi2 reactions are handled in the next off-chip scheduling step.

Algorithm 1 shows the pseudo code for final scheduling of reactions associated with off-chip species data. Based on renamed species indices, the compiler brings consecutive lines of species data into the cached region of the scratchpad. In order of reaction type, it then searches for reactions whose reactants are in either cached or uncached scratchpad memory. Such reactions are scheduled and inserted into a queue, which is used for postprocessing in subsequent steps of the compilation flow. If the eviction of a scratchpad line is needed, we heuristically choose the one least recently fetched. While searching for schedulable reactions, reactions of type Bi2 are only considered if the registers of the RU's molecules merger unit are not fully occupied or if they already contain a companion reaction with the same index. In the latter case, if the reaction gets scheduled, the register value is invalidated and the entry count is decremented.

TABLE II
EFFECT OF COMPILER OPTIMIZATIONS ON SSP PERFORMANCE.

| SSP | | Network | | Throughput [ksteps/s] | |
|---|---|---|---|---|---|
| $S$ | $R$ | $M$ | $N$ | No-opt | Opt |
| 1 | 8 | 32,768 | 4,096 | 28.4 | 48.9 (172%) |
| 1 | 8 | 65,536 | 8,192 | 10.2 | 21.4 (211%) |
| 1 | 16 | 32,768 | 4,096 | 30.8 | 43.3 (141%) |
| 1 | 16 | 65,536 | 8,192 | 10.6 | 17.0 (161%) |
| 4 | 32 | 32,768 | 4,096 | 43.5 | 44.7 (103%) |
| 4 | 32 | 65,536 | 8,192 | 14.5 | 16.7 (115%) |

To evaluate the effects of the renaming/reordering algorithm, we measured SSP throughput for simulation of synthetic colloidal aggregation models (see section V) with compiler optimizations in steps (ii) through (v) enabled or disabled. With optimizations disabled, we randomly partitioned reactions and randomly shuffled the order of both reactions and species, but we still allowed the caching feature of the scratchpad. We assumed that all species prefetching instructions are locally stored for both cases. Table II shows results for large networks with varying SPE configurations and network sizes. Throughput gains using our simple heuristics range from 103% up to 211%. There are many opportunities for further improvements. This includes boosting the currently slow compilation time by applying sophisticated search algorithms, rigorously determining which scratchpad lines to evict for better caching, applying DRAM-aware partitioning and mapping to reduce latency, and developing intelligent network analysis methods especially for those containing many Bi2 reactions.

## V. EVALUATION

We have developed a cycle-accurate simulator of our SSP design that is coupled with an external DRAM timing simulation [27] and a preliminary version of our compiler. We configured our simulator with different combinations of hardware sizes in terms of SPE and RU counts. We assumed that the SSP can accommodate a total of 4MB of on-chip memory, which matches recent mid-end GPUs [28]. For 128 RUs, this implies that the SSP can locally hold networks with up to 1024 species and reactions per RU, or around 131,000 reactions and species total. We conservatively assumed a clock frequency of 400MHz. To obtain latency numbers of all floating-point operators, we synthesized Synopsys Designware libraries using a 40nm ASIC technology. We used published performance data of other components, such as random number generators [20] and lookup-based logarithm computation [29]. In addition, we realistically modeled latencies for all data and control paths in the design. Specifically, we incorporated delay models of reorder buffers into the memory interface. Various other

TABLE III
CONFIGURED SIMULATOR PARAMETERS.

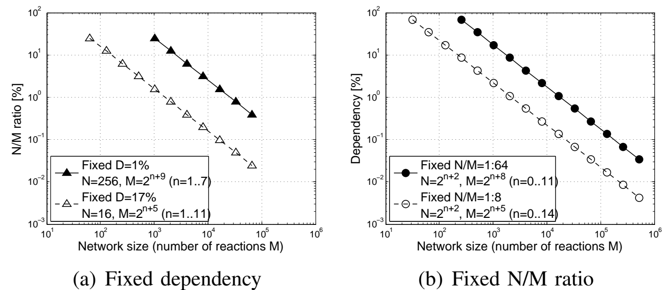
| SSP @40nm ASIC technology | |
|---|---|
| Number of cores | Up to 128 SPEs or RUs |
| Operating frequency (Main/DRAM-related) | 400MHz / 800MHz |
| RU/Reaction broadcast/Vector broadcast/Min Tree | 22 / $\ln S$ / $\ln R$ / $\ln R$ cycles |
| Reaction/Species/Population/State-change vector | 1 / 2 / 2 / 4 words/64-bit |
| Per RU reaction memory | 8kB |
| Per RU scratchpad memory (uncached/cached) | 4kB / 4kB |
| Per RU bimolecular registers | 32 entries |
| Per SPE vector cache: 16kB direct-mapped main cache, 16-entry fully assoc. victim cache | |
| Per SMU species prefetching memory | 4kB |
| DRAM request and data buffers | 32 entries each |
| **DRAMSim2** | |
| DDR3-1600 DRAM (# channels/ranks/banks) | 8 / 4 / 8 |
| DRAM data bus/Access granularity | 8B / 64B |
| Per channel bandwidth | 12.8GB/s @800MHz |
| Memory controller: out-of-order, open-page row buffer policy | |
| Queuing structure/Scheduling policy: per rank per bank / rank then bank round robin | |
| Transaction and command queues | 32 entries each |

configured parameters for our SSP simulator as well as for DRAMSim2 [27] are summarized in Table III.

To evaluate the effects of varying network characteristics on compiler and hardware performance, we created a range of artificial networks exhibiting different dependency factors and $N/M$ ratios. Artificial networks were created by hierarchically replicating a base colloidal aggregation network [10] up to $2^{16}$ times. The base network contains eight reactions and four species with a ratio of unimolecular to bimolecular reactions of 1:1 and exhibits an N/M ratio of 1:2 and a dependency factor of about 70%, i.e., on average five or six reactions are affected by every reaction firing. We controlled the N/M ratio or the dependency factor either by assigning the same original species to all replicated sets of reactions or by creating new copies of species, i.e., each reaction set having its own species. We randomly assigned rate constants of reactions and set the initial populations of all species to 10 molecules. Through this process, we synthesized artificial networks with sizes of up to 524,288 reactions and 65,536 species. Due to the way we create networks, dependency factors and the $N/M$ ratios can not be simultaneously controlled, i.e., with one parameter being fixed, the other varies to a wide extent. Fig. 6 summarizes the characteristics of various generated networks with fixed dependencies or fixed N/M ratios.

We compare SSP performance with software implementations of best-in-class SSA-CR and heap- and hashing-based NRMs. We collected software measurements using a state-of-the-art simulator (Cain1.10 [9]) running on a 24-core, 2.93GHz Intel Xeon X5670 server with 75GB main memory. For both software and hardware measurements, we simulated each network to account for a total of 100,000 time steps. Fig. 7 shows the scaling of simulator performance with network size for networks with fixed dependency $D$ or fixed $N/M$ ratio.

In Fig. 7(a) and 7(b), we can observe the impact of a (fixed) dependency on performance. While the SSP exhibits orders of magnitude better performance for strongly-coupled networks, these gains shrink with reduced network dependence. For loosely-coupled networks, SSP configurations that only aim to exploit fine-grain parallelism become similar in performance to software. However, exploiting coarse-grain SSP parallelism across independent simulations still results in an order of magnitude speedup. For networks with fixed $N/M$ ratios in Fig. 7(c) and 7(d), SSP performance scales worse than software. This is due to the exponential scaling of dependency factors with network size for the types of fixed $N/M$ networks we consider. As shown in Fig. 6(b), for networks larger than $M$=$10^4$, dependencies reach unrealistic extremes as low as $D$=0.004%. This leads to significant advantages for software-based methods that can exploit dependency, such as NRMs.



(a) Fixed dependency (b) Fixed N/M ratio

Fig. 6. Characteristics of tested synthetic networks.

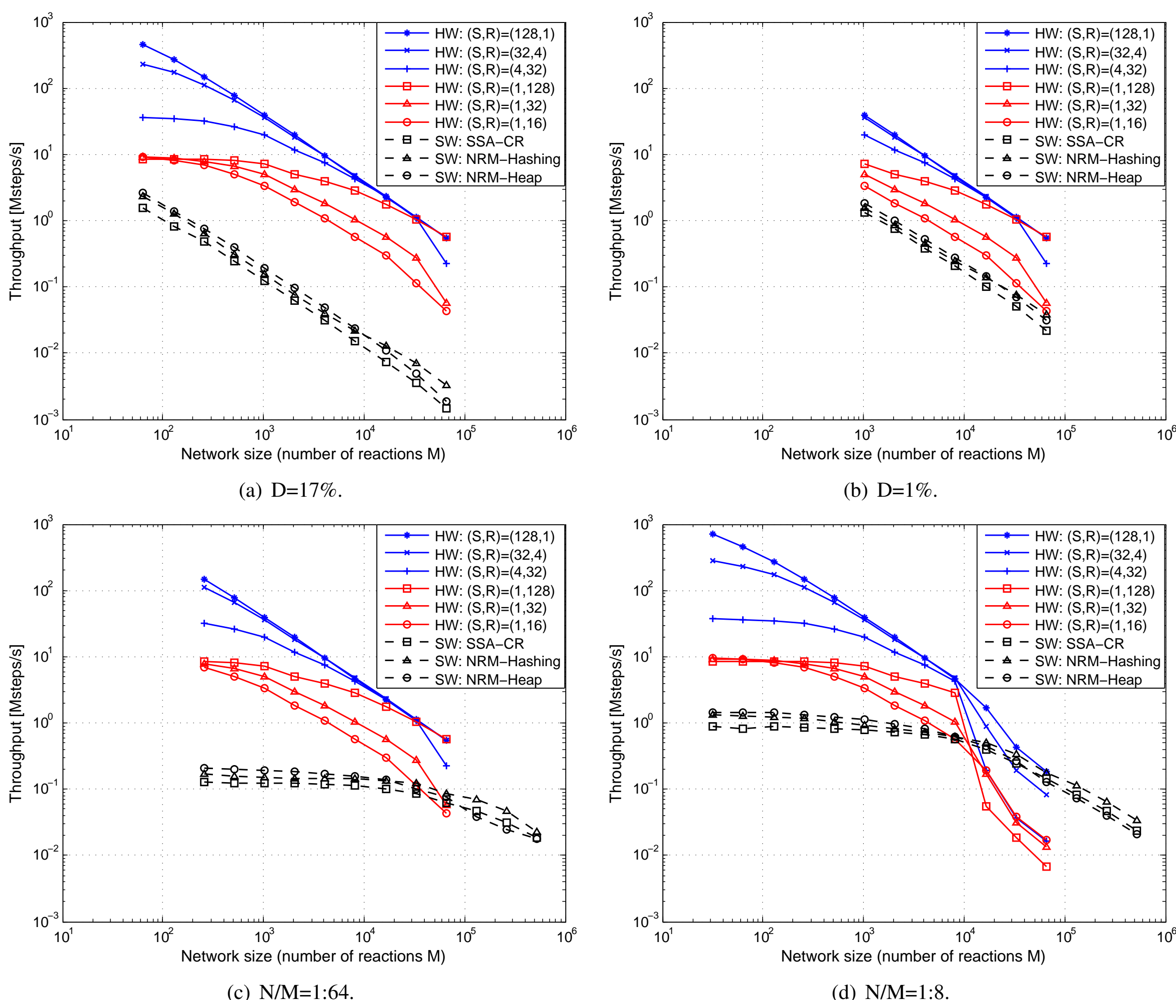
In all cases of Fig. 7, we can see how external memory interface limitations affect hardware performance. Once the amount of reaction and species data exceeds SSP-internal memory sizes and available bandwidth, performance starts to decrease. Since we can share reaction but not species traffic across SPEs, this is especially prominent for networks with high species-to-reaction ratios and for configurations with a low number of SPEs but a high number of RUs. Overall, we can see that configuring the SSP with 128 SPEs having 1 RU indicates the best performance over all configurations.

## VI. CONCLUSIONS

Stochastic simulation algorithms (SSAs) are an important tool in computational biology. In this paper, we presented the design of a stochastic simulation processor (SSP) that realizes a first reaction method (FRM) in dedicated hardware. Our design supports simulation of large-scale biochemical networks with millions of reactions and species. It exploits both fine- and coarse-grain parallelism across reactions in a network and across Monte Carlo simulations of the same network. Hardware is supported by an automated and optimizing compiler that can map standard network descriptions onto the SSP in a user-friendly manner. Results show that orders of magnitude better performance can be achieved across a wide range of network sizes and characteristics when compared to state-of-the-art software simulations running on powerful compute servers. In future work, we will incorporate SSP support for NRM-based methods that further improve performance especially for loosely-coupled networks. In addition, we plan to develop advanced compiler optimizations for aggressive renaming and reordering.

## REFERENCES

[1] X. Cai and X. Wang, “Stochastic modeling and simulation of gene networks - a review of the state-of-the-art research on stochastic simulations,” *IEEE Signal Processing Magazine*, 24(1), 2007.
[2] D. T. Gillespie, “A general method for numerically simulating the stochastic time evolution of coupled chemical reactions,” *J. Comput. Phys.*, 22(4), 1976.




(a) D=17%.

(b) D=1%.

(c) N/M=1:64.

(d) N/M=1:8.

Fig. 7. Simulated results for networks with fixed dependency factors and fixed N/M ratios.


[3] ——, "Exact stochastic simulation of coupled chemical reactions," *J. Phys. Chem.*, 81(25), 1977.

[4] A. M. Kierzek, "STOCKS: STOChastic Kinetic Simulations of biochemical systems with Gillespie algorithm," *Bioinformatics*, 18(3), 2002.

[5] D. Endy and R. Brent, "Modelling cellular behaviour," *Nature*, 409(6818), 2001.

[6] J. F. Keane *et al.*, "A compiled accelerator for biological cell signaling simulations," in *FPGA*, 2004.

[7] M. A. Gibson and J. Bruck, "Efficient exact stochastic simulation of chemical systems with many species and many channels," *J. Phys. Chem. A*, 104(9), 2000.

[8] Y. Cao *et al.*, "Efficient formulation of the stochastic simulation algorithm for chemically reacting systems," *J. Chem. Phys.*, 121(9), 2004.

[9] S. Mauch and M. Stalzer, "Efficient formulations for exact stochastic simulation of chemical systems," *TCBB*, 8(1), 2011.

[10] R. Ramaswamy *et al.*, "A new class of highly efficient exact stochastic simulation algorithms for chemical reaction networks," *J. Chem. Phys.*, 130(24), 2009.

[11] A. Slepoy *et al.*, "A constant-time kinetic monte carlo algorithm for simulation of large biochemical reaction networks," *J. Chem. Phys.*, 128(20), 2008.

[12] H. Salis *et al.*, "Multiscale Hy3S: Hybrid stochastic simulation for supercomputers," *BMC Bioinformatics*, 7(1), 2006.

[13] T. Tian and K. Burrage, "Parallel implementation of stochastic simulation for large-scale cellular processes," in *HPCASIA*, 2005.

[14] K. Burrage *et al.*, "A grid implementation of chemical kinetic simulation methods in genetic regulation," in *APAC03*, 2003.

[15] J. M. McCollum *et al.*, "Accelerating gene regulatory network modeling using grid-based simulation," *SIMULATION*, 80(4-5), 2004.

[16] H. Li and L. Petzold, "Efficient parallelization of the stochastic simulation algorithm for chemically reacting systems on the graphics processing unit," *Int. J. High Perform. Comput. Appl.*, 24(2), 2010.

[17] D. Jenkins and G. Peterson, "GPU accelerated stochastic simulation," *SAAHPC*, 2010.

[18] N. Jun-qing *et al.*, "A distributed-based stochastic simulation algorithm for large biochemical reaction networks," in *ICBBE*, 2007.

[19] M. Yoshimi *et al.*, "Stochastic simulation for biochemical reactions on FPGA," in *FPL*, 2004.

[20] ——, "An FPGA implementation of high throughput stochastic simulator for large-scale biochemical systems," in *FPL*, 2006.

[21] ——, "FPGA implementation of a data-driven stochastic biochemical simulator with the next reaction method," in *FPL*, 2007.

[22] M. Hucka *et al.*, "Evolving a lingua franca and associated software infrastructure for computational systems biology: the Systems Biology Markup Language (SBML) project," *IEE Systems Biology*, 1(1), 2004.

[23] H. Park, X. Shen, H. Vikalo, and A. Gerstlauer, "Algorithm/architecture co-design of a stochastic simulation system-on-chip," UT Austin, Tech. Rep. UT-CERC-11-01, 2011.

[24] J. M. McCollum *et al.*, "The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior," *Comput. Biol. Chem.*, 30(1), 2006.

[25] H. Li and L. Petzold, "Logarithmic direct method for discrete stochastic simulation of chemically reacting systems," UCSB, Tech. Rep., 2006.

[26] B. J. Bornstein *et al.*, "LibSBML: an API Library for SBML," *Bioinformatics*, 24(6), 2008.

[27] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, 10(1), 2011.

[28] Nvidia, "Fermi compute architecture white paper," Tech. Rep., 2009.

[29] S. Paul *et al.*, "A fast hardware approach for approximate, efficient logarithm and antilogarithm computations," *TVLSI*, 17(2), 2009.