Discrete Event Simulation of Molecular Dynamics with Configurable Logic*

Josh Model Martin C. Herbordt

Department of Electrical and Computer Engineering
Boston University; Boston, MA 02215
EMail: {jtmodel|herbordt}@bu.edu

Abstract: Molecular dynamics simulation based on discrete event simulation (DMD) is emerging as an alternative to time-step driven molecular dynamics (MD). DMD uses simplified discretized models, enabling simulations to be advanced by event, with a resulting performance increase of several orders of magnitude. Even so, DMD is compute bound. Moreover, unlike MD, causality issues make DMD difficult to scale, with $O(\sqrt{p})$ being the best so far achieved. We find that FPGAs are extremely well suited to accelerating DMD. The chaotic execution, which results in there being virtually no prediction window, is overcome with a long processing pipeline augmented with associative structures analogous to those used in CPU reorder buffers. Our primary result is a microarchitecture for DMD that processes events with a throughput equal to a small multiple of the FPGA's clock, resulting in a hundred-fold speed-up over serial implementations.

1 Introduction

Molecular dynamics (MD) simulations are a fundamental tool for gaining understanding of chemical and biological systems; its acceleration with FPGAs has rightfully received much recent attention [1, 3, 9, 12, 13, 21]. A major limitation, however, is that even when scaled to thousands of processors, simulations of time scales beyond nanoseconds is problematic; 9-12 orders of magnitude more time is needed to model many important biological phenomena, e.g., the protein association and aggregation subsequent to misfolding that is integral to many disease processes [6, 23].

An emerging alternative is molecular dynamics based on discrete event simulation, referred to as discrete molecular dynamics, or DMD [5, 6]. DMD uses simplified models: atoms as hard spheres, covalent bonds as infinite barriers, and van der Waals forces as square wells. This discretization enables simulations to be advanced by event, rather than time step. Events occur when two particles reach a discontinuity in interparticle potential. The result is simulations that are up to 10^8 to 10^9 times faster than traditional MD [6]. The simplified model can be substantially compensated for by the capability of researchers to interactively refine simulation models [25].

Even so, current DMD simulations are also compute bound, sometimes taking a month or more (e.g., [22]), although with far less resources than used for high-end MD simulations. In fact, a major problem with DMD is that, as with discrete event simulation (DES) in general [8], causality concerns make DMD difficult to scale to a significant number of processors [16]. FPGA acceleration of DMD is therefore doubly important: not only would it multiply the numbers of computational experiments or their model size or detail, it would do this many orders of magnitude more cost-effectively than could be done on a massively parallel processor, if it could be done that way at all.

What's so hard about parallelizing DMD, or more generally, parallel discrete event simulation (PDES)? Simulated events may change the system state in at least two ways: by causing new events, and by causing events currently scheduled to not occur. If these changes of state are unpredictable, as they are in DMD, then the concurrent processing of events is problematic. The basic problem is that overhead associated with bookkeeping of the parallel execution (e.g., updating event queues), is large with respect to the time to process an event. In some PDES application domains, it is possible to circumvent this by predicting a window during which event execution is "safe," or by making a similar assumption to ensure that the amount of work that may need to be undone is limited [8]. DMD, however, is chaotic: new events are unpredictable. There is no safe window [14].

*This work was supported in part by the NIH through award #RR020209-01 and facilitated by donations from Xilinx Corporation. Web: http://www.bu.edu/caadlab.

Our primary result is a microarchitecture for DMD that processes events at a small multiple of clock frequency, currently about one event per 1.5 cycles for small models (of a few thousand particles), for a resulting speed-up over serial implementations of 440×.[1] The critical factor enabling high performance is the use of broadcast buses that allow event invalidations and several insertions to all be processed in a single cycle. This allows use of a long processing pipeline with logic somewhat analogous to the reorder buffers used in contemporary CPUs; a key result is that the number of stalls is tolerable. Other important features are hierarchical event processing, which allows delayed processing for events far in the future thus enabling support for large models; precision management; and a novel priority queue structure. We currently implement hard spheres and covalent bonds – extensions to more complex force models are underway, but do not change the overall design.

2 Discrete Molecular Dynamics

2.1 DMD Models

The physical processes that lend themselves to DMD are often inherently long time-scale and amenable to simplified models. Examples are protein folding and aggregation. That these and other typical applications involve polymers contributes to the approximation model used. Rather than simulate every atom, as is done with finer models (used in MD), higher molecular structures (entire amino/nucleic acids or parts thereof) are represented as a small number of entities. These structures, often called beads, are the primary unit of simulation [19].

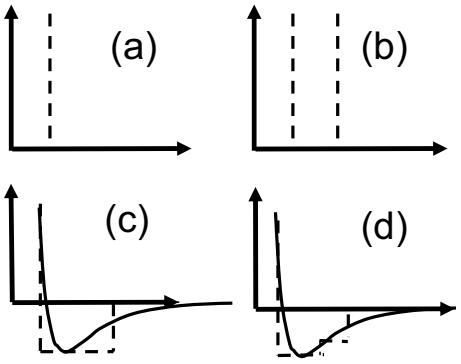

Figure 1: DMD force models.

Forces are also simplified – all interactions are folded into a square-well potential model. Figure 1a shows the infinite barrier used to model a hard sphere; Figure 1b an infinite well for a covalent bond; and Figures 1c and 1d the van der Waals model used in MD, together with square well and multi-square-well approximations, respectively.

Extensions to DMD come from solvent modeling and protein coarse-graining. Solvents are often implicit, but in larger simulations have also been modeled explicitly. Protein coarse-graining is useful when various behaviors of a protein element cannot be captured in a single bead. Several types of beads are then used to model the various desired behaviors, such as the hydrophobic nature of some amino acids [5].

2.2 DMD Overview

Overviews of DMD can be found in many standard MD references, particularly Rapaport [20]. A DMD system consists of the

- **System State**, which contains the particle characteristics such as velocity, position, time of last update, and type;
- **Event Predictor**, which transforms the particle characteristics into pairwise interactions (events); and
- **Event Processor**, which turns the events back into particle characteristics.

Events between pairs of particles are predicted by solving the ballistic equations of motion, yielding the interaction time $\tau = \frac{-b \pm \sqrt{b^2 - v^2 * (r^2 - \sigma^2)}}{v^2}$, where v is the relative velocity, r the distance, σ the diameter, and $b = v \times r$. Note that, unlike in MD, interactions conserve energy. The event processor similarly computes the post-collision motion parameters.

In serial implementations the major functional component, besides the predictor and processor, is the event calendar. This generally involves a dynamically balanced tree, leading to $O(\log N)$ processing per event insertion, although an optimization by Paul reduces this to $O(1)$ [18].

Execution progresses as follows. After initialization, the next event (processing, say, particles a and b) is popped off the calendar and processed. Then, previously predicted events involving a and b, which are now no longer valid, are removed from the calendar. Finally, new events involving a and b are predicted and inserted into the calendar.

To bound the complexity of event prediction, the simulated space is subdivided into cells (as in MD). Since there is no system-wide clock advance during which cell lists can be updated, bookkeeping is facilitated by treating cell crossings as events and processing them explicitly.

There is some trade-off as to how many predictions per particle to insert into the event calendar as a result

[1] We are currently implementing support for large models and our simulations indicate that we will be able to do this with little slowdown. This is described in Section 3.3.

of each new event. The method used here is to record only, for each bead, the next potential cell crossing plus the next bead interaction, if that would come first (see, e.g., [14]). Event execution can therefore cause at most four new events to be predicted.

3 FPGA Algorithm and Design

3.1 Overall design

The overall goal of our design is to pipeline the entire simulation described in Section 2.2 so that one event is committed every cycle. We begin with a high level description of the overall design and its components. We then address complications arising from the unpredictable event invalidations and insertions inherent in DMD.

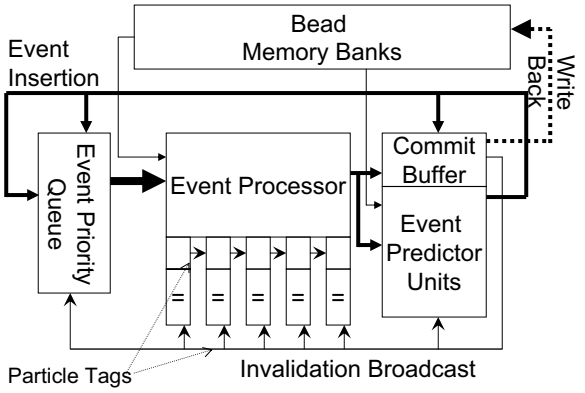

Figure 2: Block diagram of DMD simulator.

As seen in Figure 2, the FPGA DMD system consists of several hardware units. The event processor and predictor are analogues of the event processing and event prediction functions described in Section 2.2. The bead memory banks provide broadside access to bead information, such as position, velocity and time tag, for each bead in an event neighborhood (cells).

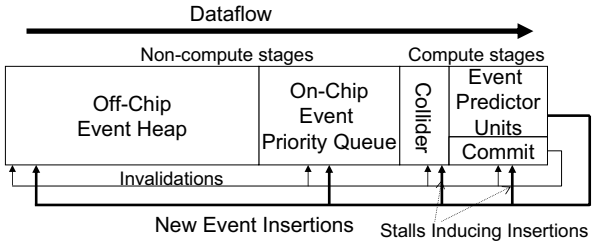

Figure 3: Conceptual diagram of DMD simulator.

The overall concept of our design is shown in Figure 3. The entire system is effectively a time-keyed priority queue. At the front of the queue are the event processor (labeled "collider") and event predictor, which together form the computation pipeline. No processing takes place in the rest of the queue. Complications result from the fact that invalidations and insertions can take place anywhere in the priority queue, including the processing parts.

The event processor transforms cell-crossings and collisions into bead updates, while the predictor generates new events from the update computations. Each new event has an associated time tag. These events are inserted into the queue at the appropriate location, based on the time tag. Insertion of events into the processing section of the queue requires special handling in order not to foul the computation in progress (see Section 3.2); insertion into the non-processing section of the queue is done using standard hardware constructs (see Section 4).

The bead memory banks store the system state. Memory is interleaved in such a way that the contents of an entire cell neighborhood can be accessed at once [26]. Committing an event involves writing the bead's new position, velocity, and time to these memories. A memory controller ensures that the collider-predictor is fed, and handles cell membership changes.

As bead updates emerge from the collider, corresponding events need to be generated. A given cell neighborhood could easily contain ten to twenty beads depending on the local density, in addition to the three candidate walls for cell crossing. Each is a potential event partner. The time of each partner must be computed. Despite all this computation, we schedule at most two events per particle prediction (four per event): the next cell crossing and the next collision, as in [14].

3.2 Complications

In an ideal DMD pipeline, the following would all take place in a single pipeline stage: events would be processed, new events predicted, invalidated events cancelled, and new events inserted into the event queue. This is not the case, however, as (currently) the event processor takes 6 and the event predictor 23 stages. Complications occur when:

Case 1. Events need to be cancelled that are already in the computation stages.
Case 2. Events need to be inserted into the computation stages.
Case 3. An event entering the event prediction processor has potentially stale information because of an event ahead that has not yet written its new state information.

The key is to retain the concept of commitment that takes place at the head of the ideal DMD pipeline. In the non-ideal DMD pipeline, just as in the ideal

pipeline: an event has committed when it emerges from the head of the priority queue (the event predictor), and when the associated invalidations and insertions have been committed (for on-chip cases this last commitment is the same as completion). The complications are addressed as follows (referring to the list above).

Case 1. This is easily accomplished in a single cycle by broadcasting the tags of the particles just processed.

Case 2. The complication here is that, although an event needs to be inserted, say, into the 3rd stage of the event processor, it has not yet gone through the first two stages. We handle this by stalling the entire pipeline, inserting the new event at the beginning of the processing pipeline, and then restarting when the inserted event has "caught up."

Case 3. As events enter the event predictor, they must check to see whether any events ahead in the queue are taking place in its own cell or its neighbors. If so, then the pipeline is stalled until that event is completed.

We now describe the effects on performance of the complications. The first two cases result from the observed uniform distributions of invalidations and insertions with respect to priority queue position.

Case 1. Event cancellations have little effect. The reasons, however, are different for the compute part and non-compute parts of the pipeline. For the non-compute part of the pipeline, we observe that the entire system is in steady state between insertions and deletions. With the priority queue design described in the next section we see that holes are quickly filled. For the compute part of the pipeline, we observe that this is a very small fraction of the event queue of a typical simulation (usually less than .05%). Moreover the effect of an unfilled hole is local: it only means that no payload will be delivered on a single cycle.

Case 2. Insertions into the compute part of the queue have more effect. Although the probability is the same as with cancellations, insertions cause a number of stalls on the order of the number of stages in the compute part of the queue. The actual number is complicated by implementation details, but still results in less than 1% loss of performance.

Case 3. Coherence stalls have similar cost as in case 2. The probability, however, is related not to the point of insertion, but rather to the ratio of the combined neighborhoods of all the events in the event predictor to the total volume of the simulation. Here is a simplified (and conservative) version of the full computation. Each event has a neighborhood of 27 cells; this times the number of events in the event predictor (23 in the current implementation) is the volume potentially affected. The total volume of the simulation in cells is a tunable parameter; $32 \times 32 \times 32$ is common. To obtain the expected performance degradation, we multiply this ratio by the number of stall cycles induced per stall (again 23) obtaining a bound of .44 stall cycles per event committed.

3.3 Handling Large Models

So far we have assumed that the entire model can fit in the on-chip portion of the priority queue (see Figure 3). While this is true for many important cases—up to several hundred beads, or several thousand particle equivalents—large models are likely to require, at least for the near future, that the substantial part of the priority queue be stored off-chip.

The question is whether this off-chip access will reduce the overall throughput to the level of software, i.e., a few hundred times slower than the on-chip throughput. Although we have not yet finished implementing the off-chip portion of the design, our preliminary examination indicates that off-chip access will not reduce performance substantially.

We begin by observing that any particular access to the off-chip priority queue will not be on the critical path that limits event processing throughput. This is because events off chip are roughly a thousand positions from the head of the queue and therefore not needed for many cycles, if ever. The question is then whether the overall throughput of off-chip processing is sufficient to feed the on-chip part. The rest of the argument runs as follows.

1. Off chip access does not entail processing an entire event; rather, only scheduling (and not event processing and prediction). Our profiling of serial reference codes indicates that, when using the classical tree data structure, scheduling consists of roughly 30% of the execution time.

2. Events advance substantially more slowly the further away they are from the head of the queue. This is because of insertions and invalidations. Our simulations indicate that the flow rate around location 1000 is less than half that at the head of the queue.

3. This still leaves a large factor to be accounted for. This is done by changing the monolithic tree to a hierarchical $O(1)$ data structure, as proposed by G. Paul [18]. His basic idea is that events likely to be used soon are inserted into a small ordered tree (20-30 elements) at the head of the queue, while the rest are placed into a coarsely ordered array of linked lists. The tree provides a sorting "buffer" wherein small errors in ordering can be corrected.

We extend this algorithm by replacing the small ordered tree with a hardware structure at the back of the on-chip part of the priority queue. By simply adding swapping capability among neighboring cells

in the queue, we enable reordering as the queue advances. The resulting design requires only that the off-chip component process four memory accesses per event advancement, i.e., every 25ns for that point in the queue. This should be easily achievable in systems with parallel access to multiple SRAM banks (e.g., [2]).

4 Implementation

In this section we sketch implementations of the primary components of the DMD simulator, accounting for the causality-based complications described in Section 3.2.

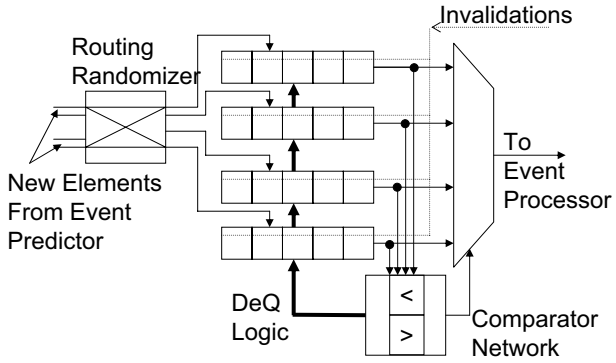

Figure 4: Four insertion event priority queue.

Event Priority Queue
The event priority queue must, on every cycle: (i) deliver the next event in time order, (ii) invalidate an unbounded number of events in the queue, and (iii) correctly insert up to four new events. This is accomplished by using broadcast mechanisms available in an FPGA. The queue is composed of four single-insertion shift register units, as seen in Figure 4. The event predictor presents two to four new elements to the routing network at each cycle. One of the 24 possible routing scenarios is pseudorandomly selected and each of the four shift register units determine the correct location to enqueue its new element. Simultaneously, dequeueing is performed by examining the heads of each of the four shift register units, and choosing the next event.

The shift register units themselves are an extension of the hardware priority queue described in [17]. Each shift register unit cell contains a time tag, the payload (bead references), a valid bit, comparators and shift control logic (see Figure 5). Since the queue is strictly ordered, the shift control logic is completely determined by the comparator results in the current and a neighboring shift register unit cell. While simultaneously dequeueing and enqueueing, the next neighbor is examined. When only enqueueing, which happens when the priority queue is not selected for dequeueing

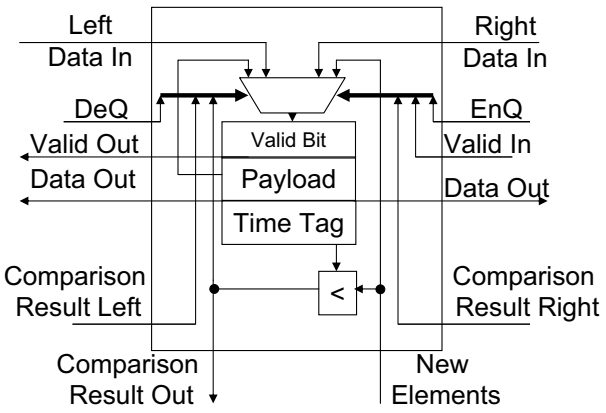

Figure 5: Single insertion, "scrunching," priority queue unit cell.

on a particular cycle, the previous neighbor is examined. The values of the tags cause the shift register control logic to perform one of four possible actions: shift forward, stay put, shift backward, or insert new element.

With up to four enqueue operations per clock cycle and only a single dequeue, the event priority queue would quickly overflow. Steady-state is maintained, however, as on average an equal number of events are inserted as removed. Invalidations are broadcast to each element of the queue every clock cycle. The issue now becomes dealing with the "holes" thus opened in the queue; these are filled through "scrunching": by looking at the valid bit of the next shift unit cell, the priority queue can use its shift control logic to remove invalid events from the queue. If the next unit is invalid, then the action is "upgraded." That is, a Shift Backwards signal becomes a Stay Put signal, and a Stay Put signal becomes a Shift Forward. Through this mechanism, new insertions do not overwhelm the queue, and holes are filled yielding payload on every cycle with high probability.

Event Processor
The event processor handles collisions and cell crossings, computing them in as few cycles as possible to avoid stalls. The implementation is a straightforward pipelining of the momentum conservation equations. The only subtlety here is that the division is replaced by a constant multiplication, as all interactions take place with a predefined radius. Parallel to the computation pipe, bead tag valid bits are propagated forward in lockstep in order to invalidate any predictions involving beads emerging from the commit buffer. In the same way, bead times are also retained to catch causality stalls. In the case of a stall, the event processor state is stored in a set of shadow registers, and the

current computation yields to the incoming event.

Event Predictor
The event predictor generates the new events to replace those that have been processed. In doing so, it provides final confirmation that updates are safe to commit, or failing that, causes a stall. Again, this is a straightforward pipelining of the ballistic equations. Various operations (division and square root) are overlapped bringing the answer together with a multiply at the end, rather than performing them serially. Similar to the event processor, a set of shadow registers maintains state in the event of a stall.

Precision Management
Precision management is not as essential to DMD as in time-step driven MD. This conclusion stems from two facts. First, energy is conserved to the limits of precision, rather than being related to the time-step size. Second, it is only essential to preserve causal orderering of events, not calculate their precise interaction times. Where precision matters most in DMD is in velocity recomputation. Here, rounding errors can result in energy being added to or removed from the simulation, and as much precision as is practical is desired.

The main datapath is 32-bits wide. For a $128\mathring{A}$ simulation box, this corresponds to a resolution of 2.9×10^{-8} Angstroms in position. Full 32-bit precision is used in the event processor to minimize velocity variation. In the event predictors, however, the parameters used for modeling covalent bond length are accurate only to 10^{-3} Angstroms. This corresponds to twelve bits (plus the five implied by the cell index), allowing us to reduce the precision used for the inputs. The output precision is returned to 32-bits over the course of its operation.

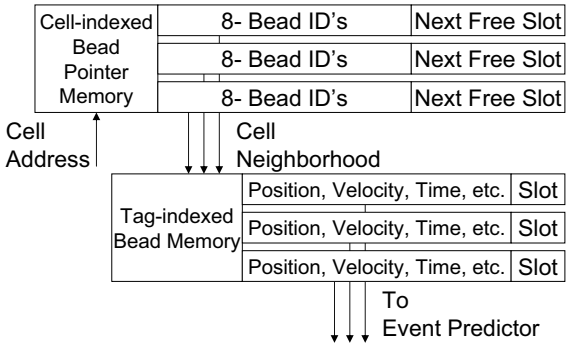

Figure 6: Bead memory block diagram.

Bead Memory
There is one type of read operation and two types of write operations the bead memory architecture must support. The read involves a neighborhood access, that, given a cell address, presents the entire neighborhood, broadside, to the event predictor. The cell address is simply the higher-order bits of the (X,Y,Z) position vector of each bead. A committed collision results in a simple write of the new position, velocity, and time to bead memory. This presents no hazard, as the FPGA RAM elements are dual-ported and can be configured to write-before-read mode.

A committed cell-crossing is more complicated. Two small auxiliary memories are required. The cell slot memory contains a bit vector for each cell indicating which slots are free. The bead slot memory stores a one-hot encoded bit vector indicating the current slot occupied by each bead. The cell slot memory values for the new and old cells are fetched as cell-crossings emerge from the commit buffer, as is the bead's current slot. This allows the bead to be placed into the new cell in a single clock cycle.

The read-mechanism works as follows. Due to the limited size of the on-chip RAM, the bead memories are arranged in a hierarchical structure as in shown in Figure 6. The bead pointer memory is interleaved by position and contains the (up to) eight pointers to beads in the addressed cell.[2] Valid bead addresses are routed to bead memories which, on the following cycle, present their contents to the event predictor.

Cell-crossing can be accomplished in a single cycle, given that the old cell slot, new cell slot, and old bead slot memory terms are available. These are fetched as the event moves through the commit buffer, and can be ready for the memory controller upon exit. The bead is inserted into the new cell by writing a pointer to the bead to the location in bead pointer memory indicated by the new cell slot. The new cell slot memory is updated to reflect its now fuller state. Removal of the bead from the old cell is accomplished simultaneously by writing a null pointer to the old-cell slot and updating the old cell slot memory to indicate its now empty state.

5 Performance and Accuracy

System Level Issues
Unlike FPGA implementations of cycle-driven MD, DMD does not appear to require substantial intervention by the host processor. This is because of the simplified models used and the favorable energy conservation properties. FPGA configurations were created in VHDL; the development flow uses Xilinx and Synplicity tools. Arithmetic retains 32-bits of precision through binary scaling except where reduced precision is safe (see Section 4).

[2]The cell size is roughly equal to the bead diameter, and as such, at most eight beads—as positioned by their centers—can be physically contained in a given cell.

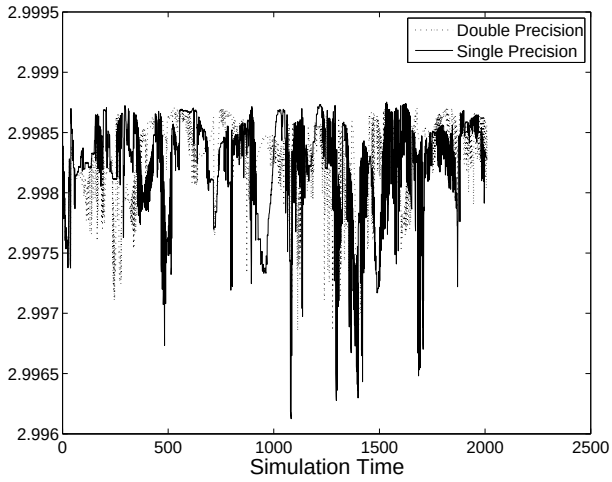

Figure 7: Total energy versus cycle for single and double precision simulations.

Validation

As with traditional MD [10], DMD is chaotic and so validation requires multiple parts. The correct working of the design is verified with a cycle accurate reference code. The operation of the cycle accurate reference code is verified with a serial reference code based on that of Rapaport [20]. The effect of reduced precision must be measured indirectly: the standard mechanism is to measure fluctuations in what should be physical invariants, such as energy. As can be seen in the snapshot shown in Figure 7, the effect of reducing precision from 53 to 24 bits does not appear to be large, so the 32 bits we are currently using could be adequate. Also, unlike MD, DMD is energy conserving to the precision of the arithmetic. In any case, precision will remain a design parameter to be varied by the computational biologist as a part of the DMD interactive experimental protocol.

Performance

Performance has two parts: throughput in events per second, and size and detail of the model that can be simulated. We first examine throughput, assuming the model fits on chip (for a discussion of off-chip access, see Section 3.3): it has three components.

1. Clock cycle time. For all implementations we have had no difficulty achieving cycle times below 10ns. Moving to the latest FPGA technology and further optimizing the implementation could improve this result.

2. Clock cycles to commit an event. With current hardware, we are able to commit events in a single cycle yielding an ideal throughput (no stalls) of one event per clock cycle.

3. Stalls per clock cycle. As discussed in Section 3.2, the number of stalls per cycle in the current design is less than .5 for a 50% reduction in performance from this source.

Combining these results, we obtain a throughput of one event per 15ns.

The model size we can fit on chip (in beads; typically multiply by 4-15 to obtain atom count) is roughly one half the size of the on-chip part of the priority queue, a result obtained from our DMD simulations. This in turn depends on the FPGA resources available and the resources required for the compute parts of the priority queue. The latter is independent of the model and FPGA size.

A sample system has been implemented on the Xilinx Virtex-II-Pro XC2VP70 -5, which we use on our Annapolis Microsystems Wildstar II-Pro. It consists of 19 Event Predictors (1920 FF per predictor) and 1 Event Processor (1922 FF). This leaves 27,774 FFs and 33,398 4-LUTs for the Event Priority Queue. This is enough logic for 150 on-chip stages. With the Xilinx V4LX200, the size of the processing section remains similar, resulting over 1000 stages fitting on-chip.

Increasing the complexity of the force model increases the size of the event processor. As this is currently a small fraction of the overall logic, the effect on model size should be modest.

Performance of Serial DMD Codes

Two serial codes were each run on two different platforms for a variety of models. The platforms were a 1.8GHz dual Opteron with 2GB RAM and compiled with GCC -O and a 2.8GHz Xeon with 2GB RAM and compiled with Microsoft Visual C++ .NET with performance optimization set to maximum. In all cases the Opteron was somewhat faster; those results are now described. The serial codes were from Rapaport [20], modified by us to handle covalent bonds, and from Donev, developed for [7]. Unlike the hardware version (modulo earlier discussion about off-chip access), serial DMD performance is dependent on model size and type. For example, simulations of small sparse models are faster than the converse. The Rapaport code achieved from 56,000 to 103,000 events per second for a range of densities and bead counts from 8,000 to 1,000. The Donev code was faster and had a substantially narrower range, achieving 143,000 to 151,000 events per second. Using the highest serial throughput numbers, we obtain a speed-up of 440× for the smaller models.

6 Discussion and Future Work

We have presented a microarchitecture for DMD and its implementation on FPGAs that obtains a substantial speed-up over serial implementations. This result is especially significant because, for reasons given throughout this paper, it will be very difficult to dupli-

cate either by replicating CPUs or with other emerging computational architectures (GPUs, Cell).

We believe this study to be the first in DMD using FPGAs. FPGAs have been used previously for other applications of DES such as traffic modeling and communication networks, and for hardware implementations of components used in DES, such as event generators and FIFOs. For a sample of this work see [4, 11, 15, 24]. These other applications have causality structures substantially different from DMD, leading to different FPGA solutions.

So far we have mostly described designs independent of whether they are implemented in ASIC or FPGA. We anticipate that configurability will be critical to successful DMD hardware architectures. As stated in the introduction, simulating phenomena over long time-scales is only one aspect of DMD use; another is its use in interactive computational experiments. More so than in typical MD usage, the DMD user designs experimental protocols around refinement of computational models, leading to the necessity of flexible DMD simulator design.

Work in progress includes finishing the off-chip priority queue and more complex force models, and performing more detailed precision studies.

Acknowledgments. We thank the anonymous reviewers for their many helpful comments.

References

[1] Alam, S., Agarwal, P., Smith, M., Vetter, J., and Caliga, D. Using FPGA devices to accelerate biomolecular simulations. *Computer 40*, 3 (2007), 66–73.

[2] Annapolis Micro Systems, Inc. *WILDSTAR II PRO for PCI*. Annapolis, MD, 2006.

[3] Azizi, N., Kuon, I., Egier, A., Darabiha, A., and Chow, P. Reconfigurable molecular dynamics simulator. In *Proc. Field Prog. Custom Computing Machines* (2004), pp. 197–206.

[4] Bumble, M., and Coraor, L. An architecture for a nondeterministic distributed simulator. *IEEE Transactions on Vehicular Technology 51*, 3 (2002), 453–471.

[5] Ding, F., and Dokholyan, N. Simple but predictive protein models. *Trends in Biotechnology 3*, 9 (2005), 450–455.

[6] Dokholyan, N. Studies of folding and misfolding using simplified models. *Current Opinion in Structural Biology 16* (2006), 79–85.

[7] Donev, A., Stillinger, F., and Torquato, S. Neighbor list collision-driven molecular dynamics simulation for nonspherical particle (parts 1 and 2). *Journal of Computational Physics 202*, 2 (2005), 737–793.

[8] Fujimoto, R. Parallel discrete event simulation. *Communications of the ACM 33*, 10 (1990), 30–53.

[9] Gu, Y., and Herbordt, M. C. FPGA-based multigrid computations for molecular dynamics simulations. In *Proc. Field Prog. Custom Computing Machines* (2007).

[10] Gu, Y., VanCourt, T., and Herbordt, M. C. Accelerating molecular dynamics simulations with configurable circuits. *IEE Proceedings on Computers and Digital Technology 153*, 3 (2006), 189–195.

[11] Keane, J., Bradley, C., and Ebeling, C. A compiled accelerator for biological cell signaling simulations. In *Proc. Field Prog. Gate Arrays* (2004).

[12] Kindratenko, V., and Pointer, D. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *Proc. Field Prog. Custom Computing Machines* (2006).

[13] Komeiji, Y., Uebayasi, M., Takata, R., Shimizu, A., Itsukashi, K., and Taiji, M. Fast and accurate molecular dynamics simulation of a protein using a special-purpose computer. *Journal of Computational Chemistry 18*, 12 (1997), 1546–1563.

[14] Lubachevsky, B. Simulating billiards: Serially and in parallel. *Int. J. Comp. in Sim. 2* (1992), 373–411.

[15] McConnell, D., and Lysaght, P. Queue simulations using dynamically reconfigurable FPGAs. In *Proc. UK Teletraphic Symposium* (1996).

[16] Miller, S., and Luding, S. Event-driven molecular dynamics in parallel. *Journal of Computational Physics 193*, 1 (2004), 306–316.

[17] Moon, S.-W., Rexford, J., and Shin, K. Scalable hardware priority queue architectures for high-speed packet switches. *IEEE Transactions on Computers TC-49*, 11 (2001), 1215–1227.

[18] Paul, G. A complexity $O(1)$ priority queue for event driven molecular dynamics simulations. *Journal of Computational Physics 221* (2006), 615–625.

[19] Rapaport, D. Molecular dynamics study of a polymer chain in solution. *J. Chemical Physics 71*, 8 (1979).

[20] Rapaport, D. *The Art of Molecular Dynamics Simulation*. Cambridge University Press, 2004.

[21] Scrofano, R., and Prasanna, V. Preliminary investigation of advanced electrostatics in molecular dynamics on reconfigurable computers. In *Supercomputing* (2006).

[22] Sharma, S., Ding, F., and Dokholyan, N. Multiscale modeling of nucleosome dynamics. *Biophysical Journal 92* (2007), 1457–1470.

[23] Snow, C., Sorin, E., Rhee, Y., and Pande, V. How well can simulation predict protein folding kinetics and thermodynamics? *Annual Review of Biophysics and Biomolecular Structure 34* (2005), 43–69.

[24] Tripp, J., Mortveit, H., Hansson, A., and Gokhale, M. Metropolitan road traffic simulation on FPGAs. In *Proc. Field Prog. Custom Computing Machines* (2005).

[25] Urbanc, B., Borreguero, J., Cruz, L., and Stanley, H. *Ab initio* discrete molecular dynamics approach to protein folding and aggregation. *Methods in Enzymology 412* (2006), 314–338.

[26] VanCourt, T., and Herbordt, M. Application-dependent memory interleaving enables high performance in FPGA-based grid computations. In *Proc. Field Prog. Logic and Applications* (2006), pp. 395–401.