An Efficient $O(1)$ Priority Queue for Large FPGA-Based Discrete Event Simulations of Molecular Dynamics[*][†]

Martin C. Herbordt Francois Kosie[‡] Josh Model[§]

Computer Architecture and Automated Design Laboratory
Department of Electrical and Computer Engineering
Boston University; Boston, MA 02215

Abstract: Molecular dynamics simulation based on discrete event simulation (DMD) is emerging as an alternative to time-step driven molecular dynamics (MD). Although DMD improves performance by several orders of magnitude, it is still compute bound. In previous work, we found that FPGAs are extremely well suited to accelerating DMD, with speed-ups of $200\times$ to $400\times$ being achieved. Large models, however, are problematic because they require that most predicted events be stored in off-chip memory, rather than on the FPGA. Here we present a solution that allows the priority queue to be extended seamlessly into off-chip memory, resulting in a throughput equal to the hardware-only priority queue, or about $30\times$ faster than the best software-only algorithm. The solution is based on the observation that—when an event is predicted to occur far in the future—not only can its processing be imprecise, but the time when the processing itself occurs can also be substantially delayed. This allows numerous optimizations and restructurings. We demonstrate the resulting design on standard hardware and present the experimental results used to tune the data structures.

1 Introduction

Molecular dynamics (MD) simulation is a fundamental tool for gaining understanding of chemical and biological systems; its acceleration with FPGAs has received much recent attention [1, 3, 8, 9, 10, 11, 20, 23]. Reductionist approaches to simulation alone, however, are insufficient for exploring a vast array of problems of interest. For example, with traditional time-step-driven MD, the computational gap between the largest published simulations and cell-level processes remains at least 12 orders of magnitude [6, 22].

In contrast, intuitive modeling is hypothesis driven and based on tailoring simplified models to the physical systems of interest [5]. Using intuitive models, simulation length and time scales can exceed those of time-step driven MD by eight or more orders of magnitude [6, 21]. The most important aspect of these physical simplifications, with respect to their effects on the mode of computation, is discretization: atoms are modeled as hard spheres, covalent bonds as hard chain links, and the van der Waals attraction as square wells. This enables simulations to be advanced by event, rather than time step: events occur when two particles reach a discontinuity in interparticle potential.

Even so, as with most simulations, discrete event simulation (DES) of molecular dynamics (DMD) is compute bound. A major problem with DMD is that, as with DES in general [7], causality concerns make DMD difficult to scale to a significant number of processors [14]. Any FPGA acceleration of DMD is therefore doubly important: not only does it multiply the numbers of computational experiments or their model size or level of detail, it does so 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.

In recent work [15] we presented an FPGA-based microarchitecture for DMD that processes events with a throughput equal to a small multiple of clock frequency, with a resulting speed-up over serial implementations of $200\times$ to $400\times$. The critical factor enabling high performance is an $O(1)$ hardware imple-

[*]This work was supported in part by the NIH through awards #R21-RR020209-01 and and #R01-RR023168-01A1, and facilitated by donations from XtremeData, Inc., SGI, and Xilinx Corporation. Web: http://www.bu.edu/caadlab.

[†]EMail: {herbordt|kosief|jtmodel}@bu.edu

[‡]Currently at Teradyne, Inc.

[§]Currently at MIT Lincoln Laboratory

mentation of common associative operations. In particular, the following standard DES/DMD operations are all performed in the same single cycle: global tag comparisons, multiple (but bounded) priority queue insertions, multiple (unbounded) priority queue invalidations, and priority queue dequeue and advance.

This event-per-cycle performance, however, depends on a pipelined hardware priority queue, which in turn appears to indicate that the entire simulator must reside on a single FPGA. While such a system is adequate for some important cases—*i.e.,* for models with up to a few thousand particles through meso-time-scales—it is also severely limited. The problem addressed here is how to extend FPGA acceleration of DMD to support large models. In the process, we also address the wider-ranging problem of creating *large efficient* $O(1)$ *priority queues*, where by "large" we mean too big to be done solely in hardware on a single chip, and by "efficient" we mean with a very small constant factor within the Big-O.

The issues are that in any realistic FPGA-based system, much of the queue will necessarily reside in off-chip memory, and that the well-known priority queue software algorithms require data structures whose basic operations often have $O(\log N)$ complexity [19]. A recent improvement yields $O(1)$ complexity [17], but remains a factor of $30\times$ slower than the hardware. The goal here is therefore to prevent the off-FPGA memory accesses, needed to support simulation of large models, from reducing the overall throughput to the level of software.

The solution is based on the observation that when an event is predicted to occur far in the future, then not only can its processing be imprecise (an idea already used by Paul [17]), but that the time when the processing itself occurs can also be substantially delayed. This allows us to optimize the data structures, reorganize the queue operations, trade off bandwidth for latency, and allow the latency of queue operations to be completely hidden. The result is that the hardware priority queue can be extended off-chip while retaining full performance. We demonstrate this on standard hardware and present the experimental results used to tune the data structures.

2 Discrete Molecular Dynamics

2.1 DMD Overview

One simplification used in DMD models is the aggregation of physical quantities such as atoms, entire residues, or even multiple residues into single simulation entities. These are often referred to as *beads*, a term originating from the simulation of polymers as "beads on a string." Forces are also simplified: all interactions are folded into step-wise potential models. Figure 1a shows the infinite barrier used to model a hard sphere; Figure 1b an infinite well for a covalent bond (hard chain); and Figures 1c and 1d the van der Waals model used in MD, together with square well and multi-square-well approximations, respectively. It is through this simplification of forces that the computation mode shifts from time-step-driven to event-driven.

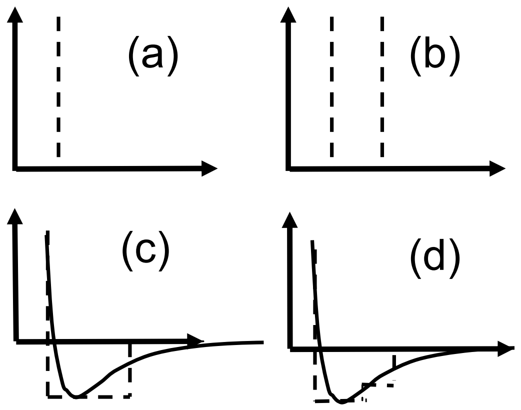

Figure 1: DMD force models: Force versus interparticle distance for a) hard spheres, b) hard spheres and hard chain link, c) simple LJ square well, and d) LJ with multiple square wells.

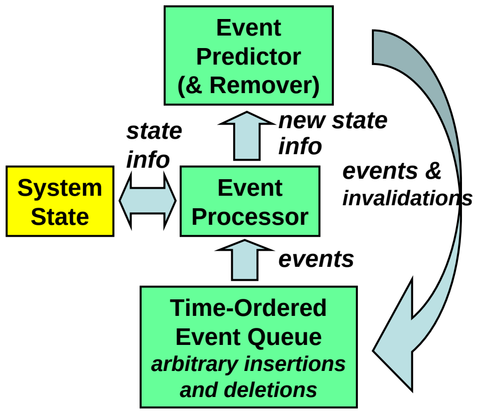

Figure 2: Block diagram of a generic discrete event simulator.

Overviews of DMD can be found in many standard MD references, particularly Rapaport [19]. A DMD system follows the standard DES configuration (see Figure 2) and consists of the

- **System State**, which contains the particle characteristics: velocity, position, time of last update, and type;
- **Event Predictor**, which transforms the particle characteristics into pairwise interactions (events);
- **Event Processor**, which turns the events back into particle characteristics; and
- **Event Priority Queue**, which holds events waiting to be processed ordered by time-stamp.

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

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. Using a scheme proposed by Lubachevsky [12], each event execution causes at most four new events to be predicted. The number of invalidations per event is not similarly bounded.

3 Priority Queue Background

3.1 Software Priority Queues

The basic operations for the priority queue are as follows: dequeue the event with the highest priority (smallest time-stamp), insert newly predicted events, and delete events in the queue that have been invalidated. A fourth operation can also be necessary: advancing, or otherwise maintaining, the queue to enable the efficient execution of the other three operations. The data structures typically are

- an array of bead records, indexed by bead ID;
- an event priority queue; and
- a series of linked lists, at least one per bead, with the elements of each (unordered) list consisting of all the events in the queue associated with that particular bead (see, *e.g.,* [19]).

Implementation of priority queues for DMD is discussed by Paul [17]; they "have for the most part been based on various types of binary trees," and "all share the property that determining the event in the queue with the smallest value requires $O(\log N)$ time [13]."

Using these structures, the basic operations are performed as follows.

1. Dequeue: The tree is often organized so that for any node the left-hand descendants are events scheduled to occur before the event at the current node, while the right-hand descendants are scheduled to occur after [19]. The event with highest priority is then the left-most leaf node. This dequeue operation is therefore either $O(1)$ or $O(\log N)$ depending on bookkeeping.

2. Insert: Since the tree is ordered by tag, insertion is $O(\log N)$.

3. Delete: When an event involving particles a and b is processed, all other events in the queue involving a and b must be invalidated and their records removed. This is done by traversing the beads' linked lists and removing events both from those lists and the priority queue. Deleting all events invalidated by a particular event is $O(1)$ on average as each bead has an average of slightly less than two events queued, independent of simulation size.

4. Advance/Maintain: Binary trees are commonly adjusted to maintain their shape. This is to prevent their (possible) degeneration into a list and so a degradation of performance from $O(\log N)$ to $O(N)$. With DMD, however, it has been shown empirically by Rapaport [18] and verified here (see Section 6), that event insertions are nearly randomly (and uniformly) distributed with respect to the events already in the queue. The tree is therefore maintained without rebalancing, although the average access depth is slightly higher than the minimum.

3.2 Paul's Algorithm

In this subsection we summarize work by G. Paul [17] which leads to a reduction in asymptotic complexity of priority queue operations from $O(\log N)$ to $O(1)$, and a performance increase of $2\times$ to $4\times$ for small and large simulations, respectively.

The observation is that most of the $O(\log N)$ complexity of the priority queue operations is derived from the continual accesses of events that are predicted to occur far in the future. The idea is to partition the priority queue into two structures. A small number of events at the head of the queue, say 20, are stored in a fully ordered binary tree (as before), while the rest of the events are stored in an ordered array of small unordered lists. To facilitate further explanation, let T_{last} be the time of the last event removed from the queue and T be the time of the event to be added to the queue. Each of these unordered lists contains exactly those events predicted to occur within its own interval of $T_i \ldots T_i + \Delta t$ where Δt is fixed for all lists. That is, the ith list contains the events predicted to occur between $(T - T_{last}) = i * \Delta t$ and $(T - T_{last}) = (i+1) * \Delta t$. The interval Δt is chosen so that the tree never contains more than a small number of events.

Also retained from before are the bead array and the per-bead linked lists of bead events.

Using these structures, the basic operations are performed as follows.
1. Dequeue: While the tree is not empty, operation is as before. If the tree is empty, a new ordered binary tree is created from the list at the head of the ordered array of lists.
2. Insert: For $(T - T_{last}) < \Delta t$, the event is inserted into the tree as before. Otherwise, the event is appended to the ith list, where $\lfloor i = (T - T_{last})/\Delta t \rfloor$.
3. Delete: Analogous to before.
4. Advance/Maintain: The array of lists is constructed as a circular array. Steady state is maintained by continuously "draining" the next list in the ordered array of lists whenever a tree is depleted.

For the number of lists to be finite there must exist a constant T_{max} such that for all T, $(T - T_{last}) < T_{max}$. In practice, most of the lists are small until they are about to be used.

Performance depends on tuning Δt. The smaller Δt, the smaller the tree at the head of the queue, but the more frequent the draining and the larger the number of lists. For serial implementations, Paul finds that choosing Δt so that the lists are generally < 20 maximizes performance. It also makes the number of lists very large, in the millions for large simulations. We describe bounding the number of lists in Section 6.

3.3 Hardware-Only Priority Queue

We briefly describe our hardware implementation of the DMD priority queue [15]. While not the subject of this paper, some background is necessary to see what we need to interface to.

To effect event-per-cycle throughput, 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, (iii) correctly insert up to four new events, and (iv) advance and maintain the queue. The queue is composed of four single-

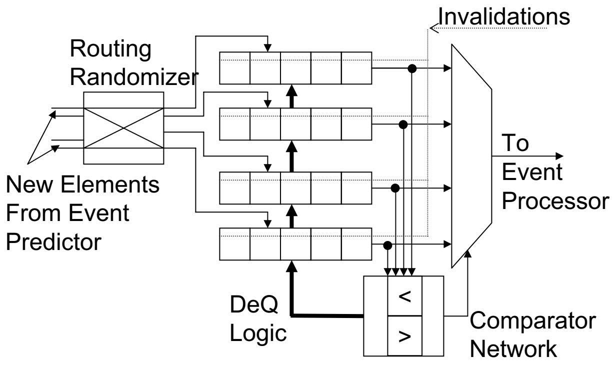

Figure 3: Four insertion event priority queue.

insertion shift register units, as seen in Figure 3. The event predictor presents two to four new elements to the routing randomizer network at each cycle. One of the 24 possible routes is pseudo-randomly selected and each of the four shift register units determine the correct location to enqueue its new element. Simultaneously, the dequeueing decision is generated 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 [16], with shift control logic completely determined by the comparator results in the current and neighboring shift register unit cells. 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 and deleted. 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," a mechanism that allows events to conditionally shift forward by two slots.

4 Off-Chip Queue Design

4.1 Constraints and Specifications

Hardware system overview
The FPGA-based system is assumed to consist of a host computer and an FPGA board. The host computer can be a PC or other microprocessor-based computational unit such as a node in a cluster or board in an SMP. The FPGA board should have an at-least moderately high-performance interface to the host. The interface is not critical, however, and a PCI connection is adequate. The FPGA board is as-

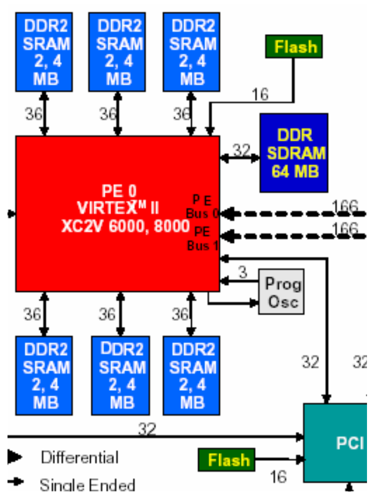

Figure 4: Partial block diagram of the Annapolis Microsystems Wildstar-II/PCI [2].

sumed to have a high-end FPGA at least of the Virtex-II/Stratix families, plus several 32-bit SRAM banks that are accessible independently and in parallel. The number and bandwidth of these banks is somewhat flexible: we compute below the minimal configuration for optimal performance and the potential performance loss of having less than that. The total SRAM size is assumed to be in the small number of MBs. FPGA boards like this are commonly available from the major vendors such as XtremeData, SGI, Annapolis Microsystems, SRC, DRC, and Nallatech (see Figure 4 for an example).

Software system overview
The DMD computation requires little partitioning. The entire application as described is self-contained. Host intervention is only required for initialization, display, periodic recalibration of temperature and/or energy, and fault recovery.

Design specification
The overall specification is to create an extension to the priority queue described in Section 3.3 that provides extra size with no performance loss. The following are the critical performance numbers.

- The cycle time is between 5ns and 10ns and is assumed to be roughly equal to the SRAM access time. While this is likely to scale with technology in the near-term, longer term changes in the ratio of on chip cycle time to off-chip SRAM access will change the resource requirements, but not the overall design.
- Events are processed at the rate of = .65 events per cycle. The ideal throughput is one event per cycle, but hazards [15] reduce the effective performance to the number indicated. The simulation of more complex force models is likely to reduce this throughput further (although not the acceleration factor).
- The event record size is 96 bits. This includes two time-stamps of 32 bits each, one each for the event time and for the time it was predicted, and up to two bead IDs of 16 bits each.
- The maximum number of events that need to be queued off-chip is 128K. This is because other design constraints limit the total number of beads to fewer than 64K beads, and because we are guaranteed to have less than two events per bead (as discussed previously). For reference, this translates into simulations of up to about 250K particles for typical intuitive models. These numbers can be increased substantially, however, without requiring any changes in the priority queue design presented here.
- The empirically determined enqueue rate for the off-chip part of the queue is 1.6 events/processed event, or about one event per cycle. Because of the uniform distribution of queue insertion position, this number is only slightly below the generation rate.
- The empirically determined dequeue rate for the off-chip part of the queue is .5 events/processed event or about .3 events/cycle. The dequeue rate is substantially less than the overall throughput because insertions and invalidations cause events to advance more slowly the farther away they are from the head of the queue.
- The total bandwidth required is therefore about 1.3 events/cycle or about 4 32-bit transfers per cycle.

The remaining problem
We have tested Paul's Algorithm by appending it to Rapaport's highly efficient DMD simulator [4, 19] and running it on a Dell Workstation with an Intel 2.5GHz Xeon E5420 Quad Core processor. For simple models, Rapaport's original DMD code requires an average of 4.4us per event of which 1.8us involves queue operations. Paul's algorithm reduces the time required for queue operations to less than 0.5us. The hardware FPGA queue, however, processes events with a throughput of one per 15ns. Assuming that the best software implementation could be interfaced directly to the hardware, this still leaves more than a factor of $33\times$ (0.5us to 15ns) to be made up.

4.2 Design

Basics
Paul's algorithm is the obvious starting point. Using this algorithm directly, however, is not practicable since too much performance would be lost in pointer following: most of the available memory bandwidth would be wasted on overhead references, rather than on transferring the data itself. The key to the design is to eliminate this overhead and so fully utilize the available bandwidth.

Motivating observation
The performance gain in Paul's algorithm is based on the observation that we should concentrate our queue computation on the events that are about to happen. This results in the execution of quick, but only partially precise, queue operations for the vast majority of events. Queue processing for an event is only completed, i.e., a total ordering of events created, just before the event is actually executed.

We take this a step further: not only can the initial queue operations be imprecise, but the time when these queue operations occur can also be imprecise. In fact the time at which a queue operation is performed can be delayed almost until the event is about to be executed. This idea allows us to apply two computational advantages of being able to create processing logic (versus software): dedicated parallel functions and trading off latency for bandwidth.

Design sketch
Since the number of memory accesses per cycle is fixed, we need to use virtually all of them for data transfer. This precludes use of pointer following, dynamic structures, and dynamic memory allocation. Instead, we rely solely on array operations.

Use of arrays in place of linked-lists leads to its own inefficiencies that must be addressed: the amount of SRAM is limited (in comparison to, say, a PC's DRAM) and likely to be a constraint on simulation size. In particular, unlike the software-only implementation, supporting millions of lists may not be viable. Rather, we need to keep the number of lists small and fixed. Also, the lists themselves must all have the same fixed size. This fixed list size (actually a bound), however, does not mean that all of the lists have the same number of elements; most are rarely more than half full. This is discussed further in Section 6. Also, because the ordering mechanism (of events in the list on draining) is much more efficient in hardware than in software, the average number of events per list can be targeted to be much larger than in software. Some other basic ideas are as follows.

- We trade off latency for bandwidth in two ways: with FIFO event queues at the output ports (the memory banks), and by periodic round-robin streaming onto the FPGA for list draining.
- No deletions of invalidated events are performed while the events are off-chip. Rather we accumulate the appropriate information on-chip and perform invalidations during draining.
- Also, we perform on-the-fly sorting of events during draining using the existing priority queues.

Data structures
We use the ordered array of unordered lists as the starting point.

- The ordered array has fixed size and resides on the FPGA. Each entry has a pointer to the beginning of its list, the list's memory bank number, and the list's current event count.
- Each list also has fixed size. Lists are interleaved among the memory banks.
- Bead memory is on-chip and represents the global system state. Each record has the bead's motion parameters, particle type, and the time-stamp of the last event in which it was involved.
- Unlike the software implementations, there are no bead-specific event lists. Rather, we use the time-stamps to process invalidations.
- To keep the number of lists fixed and moderate, the interval covered by the last list must be much less than T_{max}. As is shown in Section 6, the number of events that are "beyond current lists" (BCL) is small and can therefore be handled separately and efficiently.

Operations
Dequeue/Advance. Rather than continuously dequeueing single events, we successively drain lists by streaming them onto the chip. The timing is such that a list is virtually always being drained. Since the lists are interleaved among the memory banks, the memory bank being read rotates. Events are checked for validity as they are loaded by checking their time-stamps against those in bead memory. Events are sorted after they are loaded onto the FPGA using an extension to the hardware priority queue mechanism.

Insert. Up to four insertions can be generated per cycle but, on average, 1.5 need to go off-chip. For each event, the interval is computed and then converted into memory bank and location. The list count is incremented and the event routed to the appropriate write queue. Appended to the event is the time-stamp

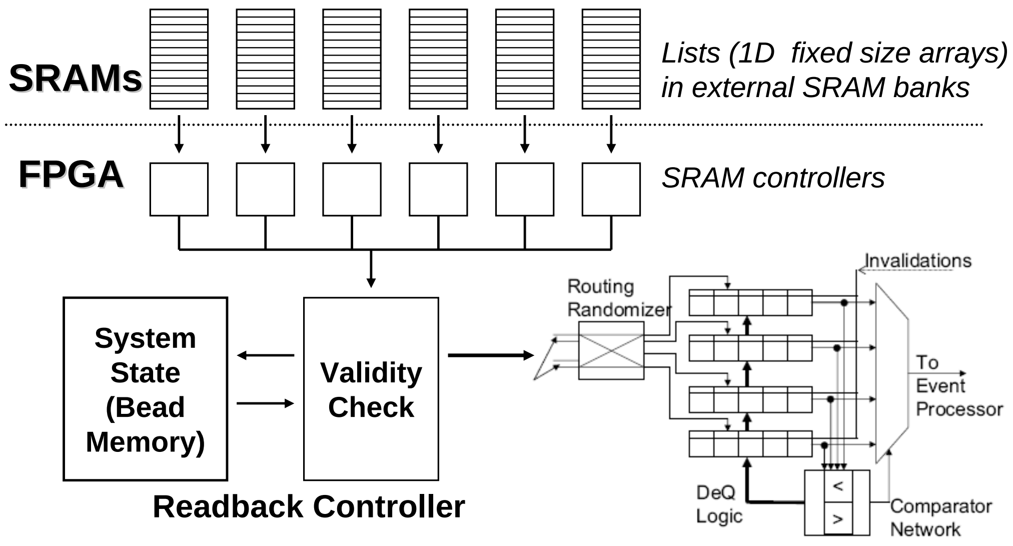

Figure 5: Block diagram of the queue advance part of the off-chip priority queue interface.

of when the event was predicted. If the memory bank is free (not being read), then the event is written.

Invalidate. Invalidations are performed as needed as lists are drained onto the FPGA. The records of both beads are checked: if an event involving either bead has occurred since the event was queued, then the event is invalidated.

5 Queue Implementation

The high-level block diagrams for a DMD off-chip controller are shown in Figures 5 and 6. The implementation is for a configuration as shown in Figure 4, but can be mapped with little change to other high-end FPGA-based systems.

Overall, there are six memory controllers for the six SRAM memory banks. Lists are distributed among the banks in round-robin fashion. Because a drain of one of these banks is almost always in progress, writes must be queued. Also, up to four events per cycle are generated, which need to have their destinations computed and then must be routed to the appropriate output queue. Some complications are the time required for enqueueing and dequeueing, and the handling of BCL events. A description of the components follows.

Readback Controller: Shown in Figure 5, it handles reading back events from the off-chip lists. Events are pulled out of the priority queue when a trigger is fired by the main on-chip logic. Puts events that have been read back into a small priority queue for validity checking, buffering, and partial sorting.

Memory (SRAM) Controllers: Shown in Figures 5 and 6, they perform the actual reads and writes to and from the memory banks. This is the only component that is platform specific.

The remaining components are shown in Figure 6.

Memory Bank Selector: Inputs events from the predictors and the BCL queue and determines the memory bank, the list, and the position to which they should be enqueued. Once the list has been computed, the bank and address are found in the off-chip list state.

Offchip List State: A global repository for list state; contains the number of the current base list (head of the circular queue) and the number of events currently in each list. Used to determine destination addresses of events to be written, and the next list to be drained. Requires the use of block RAMs.

Off-chip Router: Routes each event from the memory bank selector logic to the correct memory bank write queue. Since latency is not critical here, the router logic can be simplified by carrying it over a number of stages.

BCL Controller: Inserts events into a priority queue that were predicted to occur beyond the time covered by the current lists when they were generated. Handles deciding when to read back events from the BCL queue and forward them to the off-chip router.

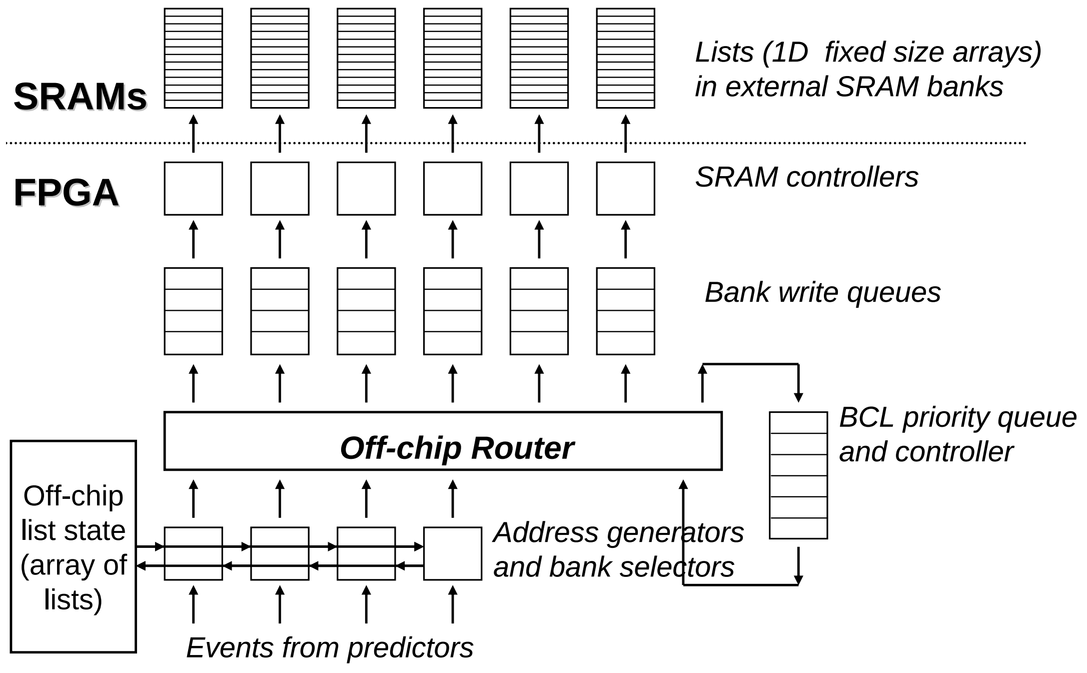
SRAMs
Lists (1D fixed size arrays) in external SRAM banks
FPGA
SRAM controllers
Bank write queues
Off-chip Router
BCL priority queue and controller
Off-chip list state (array of lists)
Address generators and bank selectors
Events from predictors

Figure 6: Block diagram of the enqueue part of the off-chip priority queue interface.

6 Performance Tuning

As with many efficient FPGA-based applications, DMD requires changes in structure from the serial version. As such, its success depends on certain critical application-dependent factors. In this section, we examine whether (i) the fixed size bound of the lists that can, with very high probability, keep them from overflowing, (ii) the number of lists necessary to keep small the number of events that are "beyond current lists," and (iii) the fraction of events that are invalidated while they are off-chip.

To determine these factors, software modeling was performed on a large number of list configurations (number of lists and list sizes). The results are now given for a design that is close to optimal for the hardware previously described. The following parameters were used:

- On-chip priority queue holds 1000 events
- 512 off-chip lists
- Scaling factor of 50 (see [17]) for a target number of events per list of 128 at draining
- List size (maximum possible number of events in any list) = 256
- Initial transient of 10,000 cycles

Distribution of maximum event count per list
Figure 7 shows a histogram of the number of events in the first off-chip list prior to that list being drained onto

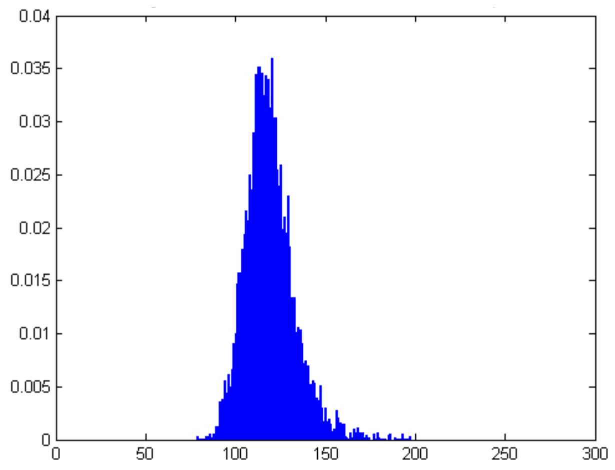

Figure 7: Histogram showing distribution of events per list for list about to be drained.

the FPGA. The distribution is tightly grouped around the target size and has a strict upper bound. That is, in our experiments, no list ever contained 200 or more events. For these physical simulations, a list size of 256 is sufficient with high probability. For different physical simulations, the average list size varies, but the variance of event counts remains small.

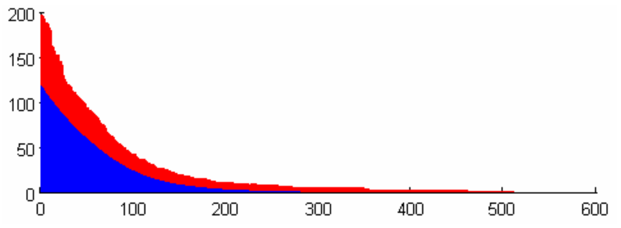

Figure 8: Graph showing average and maximum list sizes versus list number (distance from the list about to be drained).

Distribution of events through lists
Figure 8 shows the maximum and average number of events in all the lists as a function of distance of the list from the current time. Again, the distribution decreases quickly so that only a very small number of events is "beyond current lists." We also measured the number of events in the BCL queue for this configuration and found that a BCL queue size greater than 10 was never needed. We therefore conclude that a BCL queue size of 20 is sufficient, with high probability.

Proportion of events invalidated
For the invalidation-during-drain scheme to work, a large fraction of events must still be valid (not invalidated) when they are brought on-chip. For example, if 90% of the events have been invalidated, then that fraction of the transfer bandwidth will have been wasted. We find that from 58% to 62% of events have been invalidated, depending on the on-chip queue size. This means that the actual amount of "drain" bandwidth must be .75 events/cycle. This increases the total bandwidth from about 1.3 events/cycle to 1.75 events/cycle and the total bandwidth required from about 128 bits/cycle to about 170 bits/cycle. This translates to keeping our 6-bank design running at 88% capacity.

7 Discussion and Future Work

We have presented a priority queue design that allows for the seemless extension of the hardware queue into off-chip memory. Its significance is first that this allows high-acceleration, FPGA-based, DMD to work for models with sizes into the hundreds of thousands of particles. Further significance is that this design should work just as well for other applications of DES, and for other places where large priority queues are needed.

Several more optimizations are possible. One is to be more aggressive with number of lists and list size. This would make faults such as list overflow a regular occurrence. As long as their frequency remains at, say, less than 1 per ten million events, then implementing an external fault recovery mechanism could be cost-effective. For example, overflow detection logic could easily be added to the queues. In these rare cases events could be queued in neighboring lists since the hardware priority queue is tolerant of partially ordered insertions. Another optimization is to have multiple list sizes. As can be seen in Figure 8, only the last hundred lists are likely to have more than 50 elements. This optimization also requires another level of intervention, but would certainly enable more efficient storage.

Overall, our work in DMD continues in supporting more complex force models.

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

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 *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines* (2004), pp. 197–206.

[4] Dean, T. Parallelizing discrete molecular dynamics simulations. Master's thesis, Department of Electrical and Computer Engineering, Boston University, 2008.

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

[8] Gu, Y., and Herbordt, M. FPGA-based multigrid computations for molecular dynamics simulations.

In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines* (2007), pp. 117–126.

[9] Gu, Y., VanCourt, T., and Herbordt, M. Improved interpolation and system integration for FPGA-based molecular dynamics simulations. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications* (2006), pp. 21–28.

[10] Kindratenko, V., and Pointer, D. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines* (2006).

[11] 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.

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

[13] Marin, M., and Cordero, P. An empirical assessment of priority queeus in event-driven molecular dynamics simulation. *Computer Physics Communications 92* (1995), 214–224.

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

[15] Model, J., and Herbordt, M. Discrete event simulation of molecular dynamics with configurable logic. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications* (2007), pp. 151–158.

[16] 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.

[17] Paul, G. A complexity $O(1)$ priority queue for event driven molecular dynamics simulations. *Journal of Computational Physics 221* (2007), 615–625.

[18] Rapaport, D. The event scheduling problem in molecular dynamics simulation. *Journal of Computational Physics 34* (1980), 184–201.

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

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

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

[22] 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.

[23] Villareal, J., Cortes, J., and Najjar, W. Compiled code acceleration of NAMD on FPGAs. In *Proceedings of the Reconfigurable Systems Summer Institute* (2007).