FPGA Acceleration of Discrete Molecular Dynamics Simulation

Joshua Model

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

BOSTON UNIVERSITY

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

FPGA Acceleration of Discrete Molecular Dynamics Simulation

by

Joshua Model

B.S.E., Princeton University, 2001

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2007

Approved by

First Reader

Martin Herbordt, Ph.D.
Professor of Electrical and Computer Engineering

Second Reader

Wei Qin, Ph.D.
Professor of Electrical and Computer Engineering

Third Reader

Alexander Taubin, Ph.D.
Professor of Electreical and Computer Engineering

FPGA Acceleration of Discrete Molecular Dynamics Simulation

Joshua Model

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, DMD has not been shown to be scalable. 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 at a small multiple of the FPGA's clock, resulting in a substantial speed-up over serial implementations.

Contents

1 Introduction **1**

1.1 What is Molecular Dynamics? 1

1.2 Discrete Molecular Dynamics 2

1.3 DMD implementations and difficulties 2

2 Background **6**

2.1 Brief History of DMD and Biomolecules 6

2.2 DMD Models 7

2.3 DMD Overview 9

2.3.1 Prediction 9

2.3.2 Processing 11

2.3.3 Event Calendar 11

2.3.4 Cell Subdivision 12

3 FPGA Algorithm and Design **15**

3.1 Overall design 15

3.2 Complications 17

3.3 Handling Large Models 19

4 Implementation **21**

4.1 Event Priority Queue 21

4.2 Event Processor 23

4.3 Event Predictor 24

4.4 Precision Management 25
4.5 Bead Memory 26

5 Performance and Accuracy **28**

5.1 System Level Issues 28
5.2 Validation 28
5.3 Performance 30

6 Conclusion **34**

6.1 Discussion and Future Work 34

References **36**

List of Figures

1·1 Event Propagation Example 3

2·1 DMD Force Models 7

2·2 Two Dimensional Event Prediction 10

2·3 2D Cell Subdivision 12

2·4 2D Cell Crossing 13

3·1 Block Diagram of DMD Simulator 15

3·2 Conceptual Diagram of DMD Simulator 16

4·1 Four Insertion Event Priority Queue 21

4·2 Single Insertion, "Scrunching" Priority Queue Unit Cell 22

4·3 Event Processor Block Diagram 24

4·4 Event Predictor Block Diagram 25

4·5 Bead Memory Block Diagram 26

5·1 Total Energy vs. Cycle for Single and Double Precision Aimulations . . 29

5·2 Histogram of Queue Insertion Position 30

5·3 Radial Distribution Function for Rapaport and Hardware simulator Hard-Spheres 31

List of Abbreviations

ASIC	.....	Application Specific Integrated Circuit
CPU	.....	Central Processing Unit
DES	.....	Discrete Event Simulation
DMD	.....	Discrete Molecular Dynamics
FPGA	.....	Field-Programmable Gate Array
FIFO	.....	First-in First-out
GPU	.....	Graphical Processing Unit
MD	.....	Molecular Dynamics
PDES	.....	Parallel Discrete Event Simulation
RAM	.....	Random Access Memory
SRAM	.....	Static Random Access Memory

Chapter 1

Introduction

1.1 What is Molecular Dynamics?

Molecular Dynamics (MD) is a set of techniques used to integrate the equations of motion of an N-body system. Its heritage is based in Newton's laws of motion, and in most formulations, completely rests in classical mechanics. Traditionally, it is a fixed-timestep based technique in which the forces on each body are computed, and then applied to update the velocity and position. In chemical and biological contexts, the forces consist of van der Waals, long-range interactions and Coulombic, short range interactions. The long range interactions have an O(N^2) computational complexity in the number of bodies, however standard techniques such as neighbor-lists and cell-division exist to reduce this. Even so, obtaining simulation times beyond 100 ns is difficult, without supercomputer support. Several standard implementations of MD exist: NAMD, GROMACS and Protomol, being among the more notable.

Simulations based on MD are a fundamental tool for gaining understanding of chemical and biological systems; its acceleration with FPGAs has rightfully received much recent attention (1; 4; 13; 16; 17; 30). 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 (9; 34).

1.2 Discrete Molecular Dynamics

An emerging alternative is molecular dynamics based on discrete event simulation, referred to as discrete molecular dynamics, or DMD (7; 9). 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 occurs when two particles reach a discontinuity in interparticle potential. The result is simulations that are 10^8 to 10^9 times faster than traditional MD (9). The simplified model can be substantially compensated for by the capability of researchers to interactively refine simulation models (36). In addition, these simplifications of molecular behavior, such as those proposed by Go and Abe [Cite],can be arrived at emprically, rather than from first principles. This behavioral modeling allows for exploration of much larger, coarser-grained systems

Even so, current DMD simulations are also compute bound, sometimes taking a month or more (e.g., (31)), 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 (12), causality concerns make DMD difficult to scale to a significant number of processors (24). 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.

1.3 DMD implementations and difficulties

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 much larger than 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) (12). DMD, however, is chaotic: new events are unpredictable. There is no safe window (19).

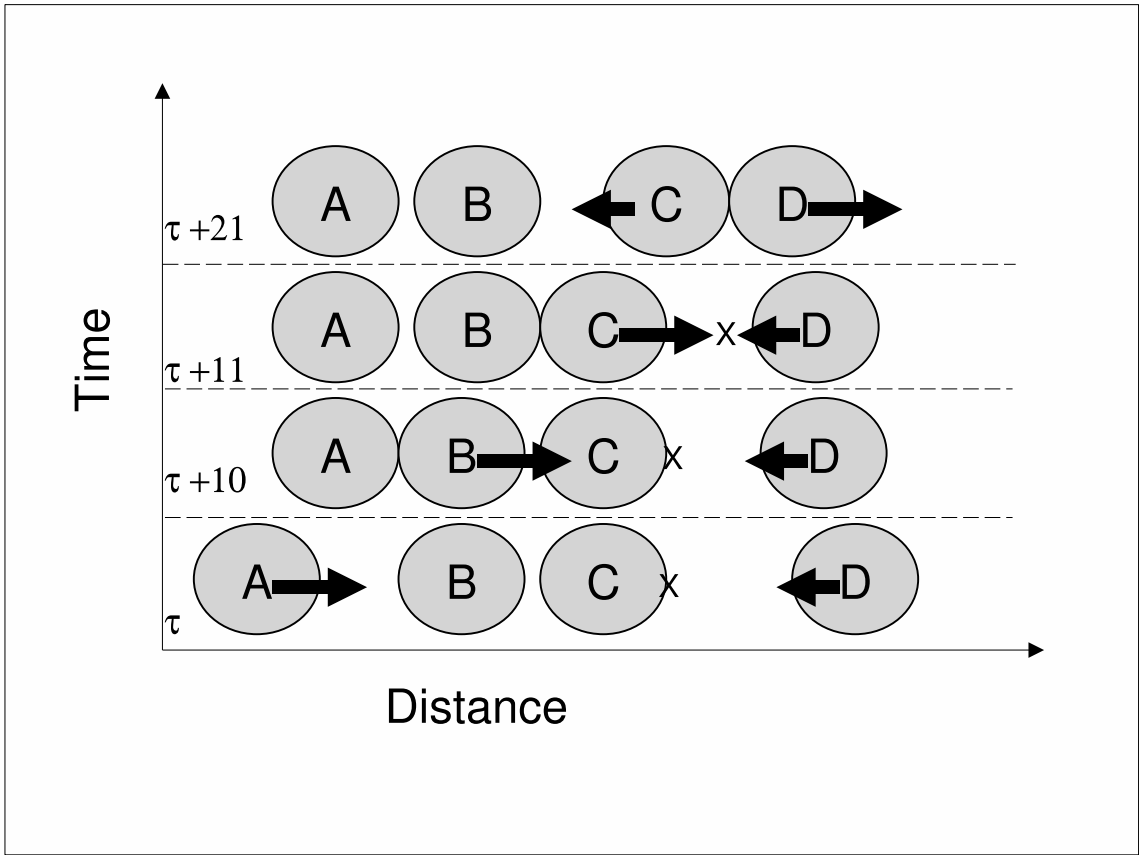

Figure 1·1: Event Propagation Example

Following (19), an example of how events can propagate without unbounded velocity

can be seen in 1·1. Consider the four discs, A,B,C and D, moving in one dimension. Their velocities are are proportional to the arrow sizes, and they all are assumed to have unit mass and diameter. The 'X' indicates the predicted location of the collision between discs C and D. Collisions are completely elastic, so when discs A and B collide at time $\tau = 10$, all of A's momentum is transferred to B. The next collision, between discs B and C, occurs soon after, as B's velocity is large and the distance between B and C is small. The result of this collison, at time $\tau = 11$, is that the 'X' is moved. Thus the A and B collision propagates its effects to C and D in a single time unit. As B and C's initial separation approaches zero, the time for the A and B collision to propagate its results to C and D also approaches zero, resulting in an unbounded event propagation velocity. This example may easily be extended to 3-space, and for sparse-gas type simulations this may be viewed as a relatively rare, pathological scenario. However, for dense simulations, or for simulatin chains of molecules, the situation described above is common.

There have been two broad strategies to parallelizing PDES, *Conservative* and *Optimistic*. The *Optimistic* approach lets each processor run its portion of the simulation until a causality violation is detected. At this point, the simulation is "rolled back" to a safe execution point, the violation corrected, and the simulation proceeds. For example, referring again to Figure 1·1, assume discs A and B are owned by processor 1, and discs C and D by processor 2 time $t\tau$. The A - B collision, and the initial C - D collision would both be processed independently. When B and C collide, at time $\tau = 11$, processors 1 and processor 2 would need to communicate the positions and velocities of those discs. The outcome would lead processor 2 to undo the original C - D collision and predict a new one for those discs.

Facilitating the ability to "undo" collisions requires saving the system state at intervals. Tighter intervals result in fewer events being undone, but memory requirements

grow rapidly. Particularly for chaotic simulations,where local causality violations can quickly become global, and a memory limited platform, this is not an attractive option. Using this approach, (24) were only able to obtain a $\sqrt{P}$ speedup.

The *Conservative* approach guarantees that causality violations will not occur, but requires that the simulation have predictability (6). This would initially rule out this approach for DMD. However, (19) and (20) describe a technique to artificially bound the event propagation velocity by creating border regions around each processor's domain where information is shared with neighboring processors. In the case that the bound is violated, the simulation stalls, resolves the violation serially, and continues. There is a dilemma involved in this technique, though. If the bound is very high, the shared information region approaches or exceeds the volume of the private regions, consuming work in communication. If the bound is low, violations become more and more frequent. (20)'s implemention also obtained $\sqrt{P}$ acceleration.

This difficulty in parallelization has led us to hew close to the serial algorithm when designing the accelerator. 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 clocks for small models (of a few thousand particles), for a resulting speed-up over serial implentations of 440×. 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, allowing processing delays 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 – extensions to production force models are underway and do not change the overall design.

Chapter 2

Background

2.1 Brief History of DMD and Biomolecules

DMD was one of the first approaches taken to simulating molecular motion in (2). Much of the early work was of was devoted to corroborating DMD techniques against analytic conjectures or more well understood Monte Carlo methods. The difficulty in parallelizing the algorithm, mentioned above, swung the pendulum towards the more scalable, timestep-driven methods. Indeed the first simulation of a biomolecule molecule was via a timestep-driven technique (22) in 1977. However, about this time, (27) and (11), published results for hard-sphere fluid and chain simulations. Of particular note (28) provided a detailed example of the 'Calendar Queue', a complex data structure to efficiently handle event scheduling. Research proceeded, often in the context of parallel and distributed simulation, but was met with limited success, as noted above. Lubachevsky pioneered the 'Bounded Lag' conservative parallel billiards simulation (18), and reduced the memory requirements of Rapaport's calendar queue by proposing lazy event evaluation(19). (21) made incremental improvements to the parallel algorithm. By the mid-to-late 1990's, though, serial processing power risen to the point where DMD simulations of significant biomolecules, or simulants began to be feasible. Smith, Dokhoylan, and Zhou all began work ((33),(8) , (38)) which to this day continues to provide insight into Amyloid-β diseases and gene expression to name two applications. While FPGAs have been used as accelerators for some discrete event

simulation applications((5), (35)) as well as traditional MD ((13)), to our knowledge no attempt has been made to accelerate DMD via FPGA.

2.2 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 (27).

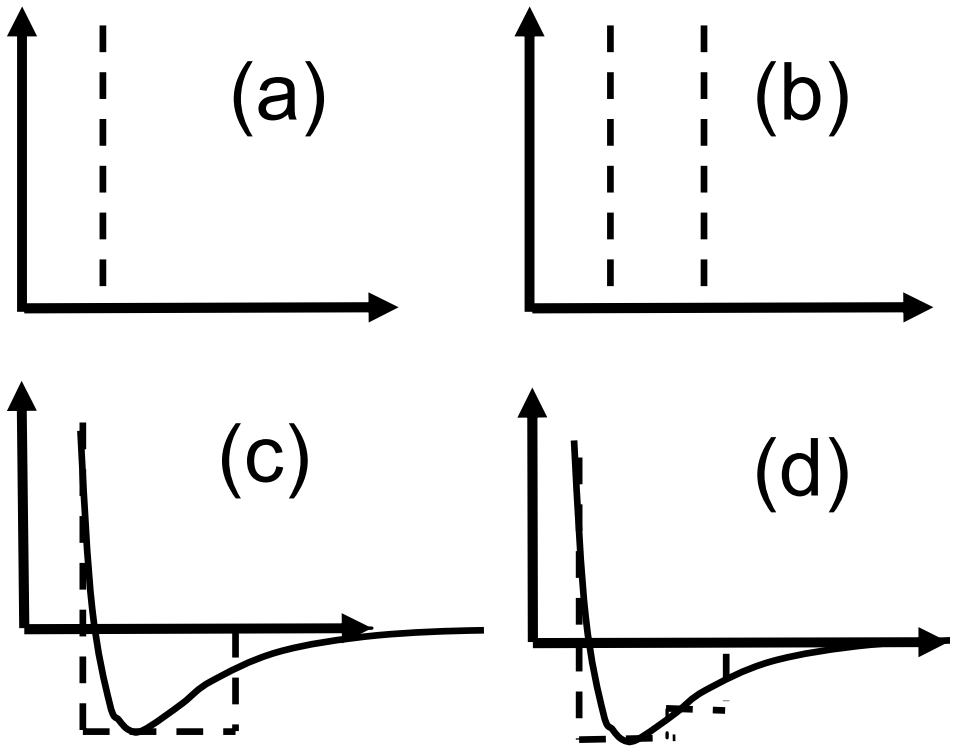

Figure 2·1: DMD Force Models

Forces are also simplified – all interactions are folded into a square-well potential model. Figure 2·1(a) shows the infinite barrier used to model a hard sphere; Figure 2·1(b) an infinite well for a covalent bond; and Figure 2·1(c) and (d) the van der Waals model used in MD, together with square well and multi-square-well approximations, respectively. The continuous line in Figure 2·1(c) and (d) represents the long range force which we are simplifying.

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 (7).

Of course, the chosen potential models have an enormous impact on the simulation results. DMD follows the "garbage-in, garbage out" principle as much as any other simulation. The potentials may be tied to reality through a number of techniques. (31) uses the mean separtations and fluctuations determined from the crystal structure of proteins to set the interaction potentials distances. Potential well depths are determined through known or experimentally measured relationships between the beads under examination. For example, following the Go-model, beads which are in contact when the protein is folded, 'native contacts', are encouraged during simulation via a potential well. Beads which are not in contact, 'non-native contacts' are penalized by a potential wall. Alternatively, the known ratio of interaction potentials between, for example, A-T and C-G nucleotide base-pairs is used to set the simulation interaction potentials. (36) takes an 'ab initio' approach which removes the dependancy on crystallographic measurements, and describes bonds and interactions analytically.

2.3 DMD Overview

Overviews of DMD can be found in many standard MD references, particularly Rapaport (29). A DMD system consists of the

- **System State**, which contains the particle characteristics such as velocity, position, time of last update, and type;
- **Predictor**, which transforms the particle characteristics into pairwise interactions (events) to be inserted into the Queue;
- **Event Processor**, which turns the events back into particle characteristics;
- **Event Calendar or Event Priority Queue**, which maintains an ordered list of events to present to the Processor.

2.3.1 Prediction

All collsion prediction involves solving the ballistic equations of motion. Following the discussion in (2) and using the notation akin to (29), Figure 2·2 shows a two dimensional example, but the equations hold for 3-space, as well.

Assume that discs A and B are both diameter σ, with initial positions R_A and R_B, and initial velocities V_A and V_B. We wish to find the time, τ, at which disc A and disc B are separated by a distance σ. R_A, R_B,V_A, V_B, dR, and dV are all vector quantities, while b,τ, and σ are scalars. So we have,

$$||(R_A + V_A \times \tau) - (R_B + V_B \times \tau)|| = \sigma \tag{2.1}$$

$$||(R_A - R_B) + \tau(V_A - V_B)|| = \sigma \tag{2.2}$$

Replacing $(R_A - R_B)$ with dR, $(V_A - V_B)$ with dV, $dV \cdot dR$ with b and squaring both sides, we have

$$dR^2 + 2\tau b + \tau dV^2 = \sigma^2 \tag{2.3}$$

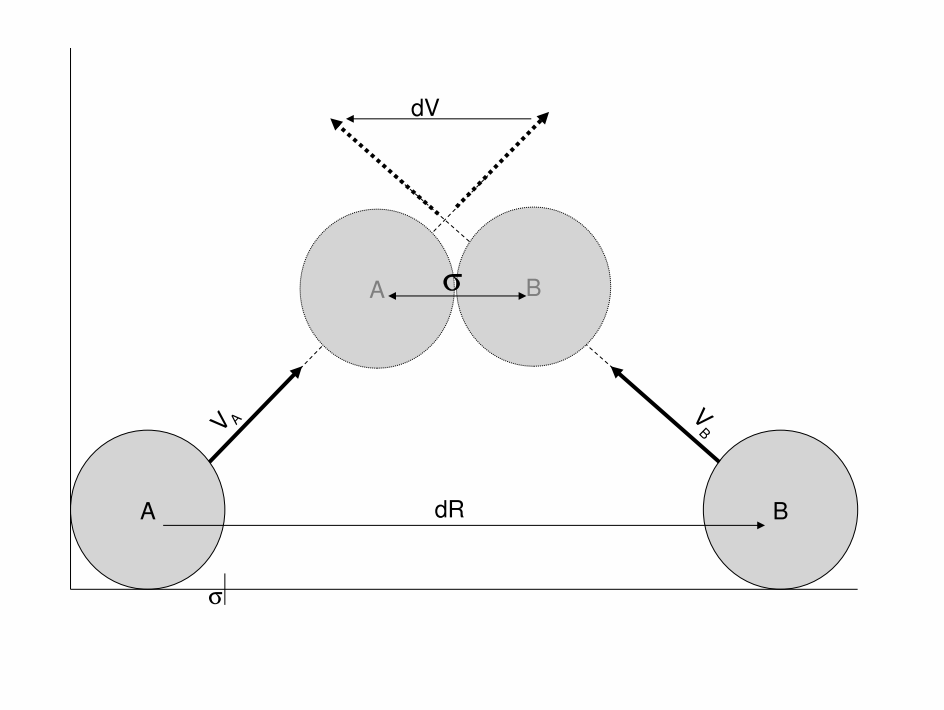

Figure 2·2: Two Dimensional Event Prediction

using the quadratic formula to solve for τ gives us

$$\tau = \frac{-b \pm \sqrt{b^2 - dV^2(dR^2 - \sigma^2)}}{dV^2} \tag{2.4}$$

for predictions of hard-sphere collisions. The bonded scenario, Figure 2·1 (b), can be handled similarly, the major difference being that the relative velocities of the two beads in question must be considered to determine if the event is to occur at the maximum or minimum bead separation. Handling well and wall potentials, as in Figure 2·1 (c) and (d), adds a significant layer of complexity to prediction. Specifically, for each prediction, it must be determined if the interacting beads have sufficient kinetic energy to leave a well. Adding this to our model represents an obvious extension to this work.

2.3.2 Processing

In DMD collision processing, unlike in continuous MD, hard-sphere or bonded interactions (Figure 2·1 (a) and (b)) conserve energy and momentum. The change in velocities, for beads of equal mass, is thus,

$$\Delta V = \frac{\pm b}{dR^2} \cdot dR \tag{2.5}$$

with dR, in this case, representing the interaction distance. In the case of a hard-sphere collision, $dR = \sigma$. For bonded beads, the maximum and minimum separations determine interaction distances, as well as the sign of the velocity change.

When considering well or wall potentials, (Figure 2·1 (c) and (d)), an interaction may occur at any of the set of discontinuities. Here, kinetic energy is not conserved, but the change in potential energy one of a few constants defined by the well depth. So the new velocities may still be calculated as,

$$\Delta V = \frac{-b \pm \sqrt{b^2 - 4dR^2(\frac{\Delta\mu}{m})}}{2dR^2} \cdot dR \tag{2.6}$$

where $\Delta\mu$ is the change in energy, and m is mass, which would allow us to discard the unit mass assumption, at the cost of a more complex bead data structure. (29).

2.3.3 Event Calendar

In serial implementations the major functional component, besides the predictor and processor, is the event calendar. This is a dynamically balanced tree, leading to $O(\log N)$ processing per event insertion. 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. There is some trade-off as to how many predictions per particle to insert into the event calendar as a result 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 (19) for details on this issue.

2.3.4 Cell Subdivision

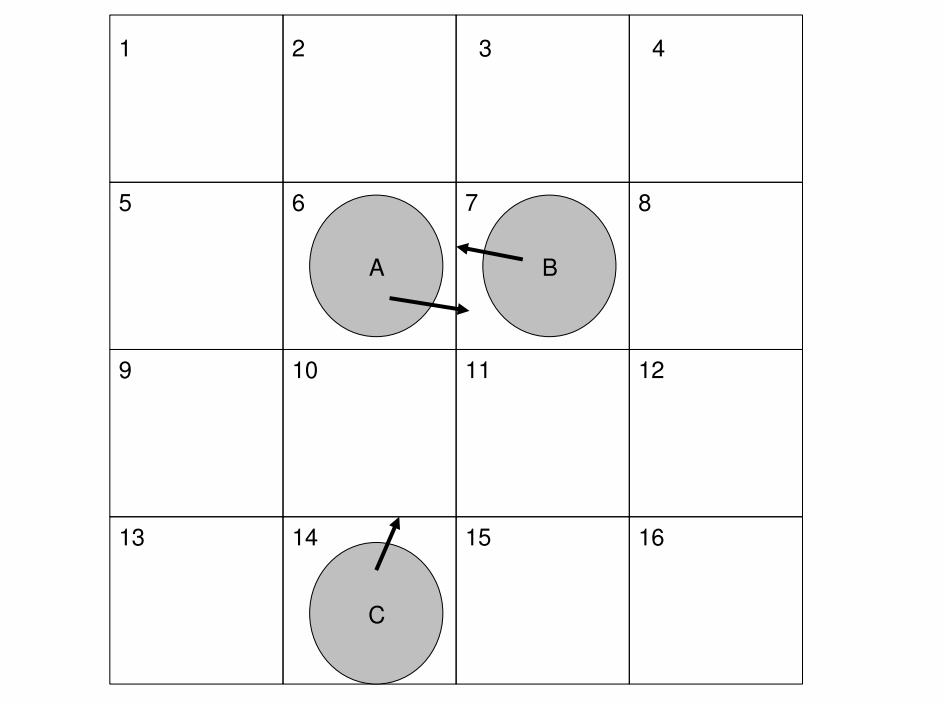

Figure 2·3: 2D Cell Subdivision

To bound the complexity of event prediction, the simulated space is subdivided into cells (as in MD) (2). Following the presentation of (32), Figure 2·3 demonstrates how cell subdivision reduces the workload. Without cell subdivision, Bead A would need to examine both Bead B and Bead C in order to determine its next collision. With cells, Bead A only needs to retrieve the beads in the neighboring cells, in this case just Bead B. Since there is no system-wide clock advance during which cell lists can be updated,

bookkeeping is done by treating cell crossings as events and predicting and processing them explicitly. In this example, this will ensure that Bead C crosses from cell 14 to cell 10 before any C-A interaction is considered. Of course, in three dimensions, the neighborhood size is twenty-seven, as opposed to the size nine neighborhood in Figure 2·3. The cell crossings are predicted and processed as collisions with virtual beads, as

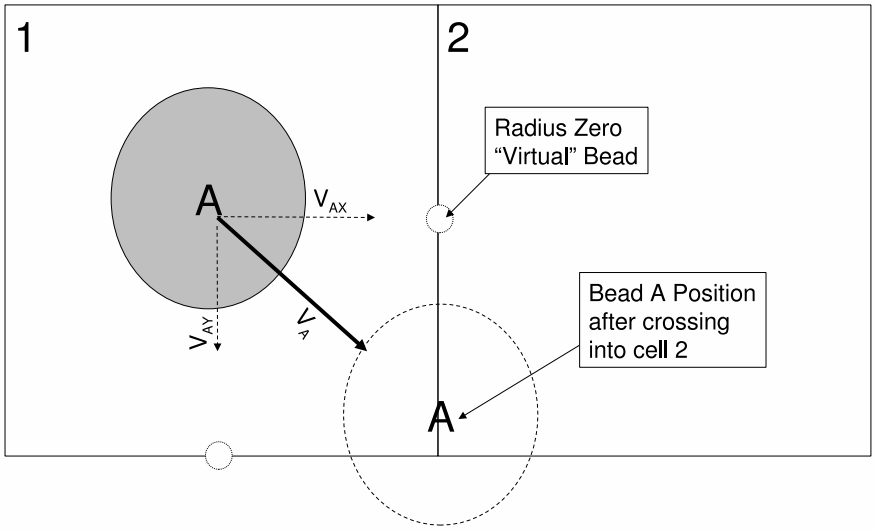

Figure 2·4: 2D Cell Crossing

seen in Figure 2·4. Once the sign of the beads velocity has been determined, there are two cell sides (or three cell faces, in three dimensions), that are candidates for the next cell crossing. A "virtual bead", velocity zero is placed at the cell edge, with the other coordinate (or other two, in three dimensions) corresponding to Bead A's position. Prediction of the time of the crossing event only considers the component of Bead A's velocity corresponding to the candidate cell edge, and occurs at an interaction

distance,σ, of zero. This reduces equation 2.4, for the X-axis cell crossing to simply:

$$\tau = \frac{dR}{dV_{AX}} \tag{2.7}$$

where dR is the distance, between Bead A and the virtual Bead. Prediction for the Y and Z axes proceeds identically. Processing a cell crossing only requires advancing Bead A along its trajectory by τ time units, as seen in the figure. It is a matter of convention whether the border between cell 1 and cell 2 belongs to cell 1 or cell 2, and will not affect the simulation so long as the convention is consistently maintained.

Chapter 3

FPGA Algorithm and Design

3.1 Overall design

The overall goal of our design is to pipeline the entire simulation described in Section 2.3 so that one event is processed per 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.

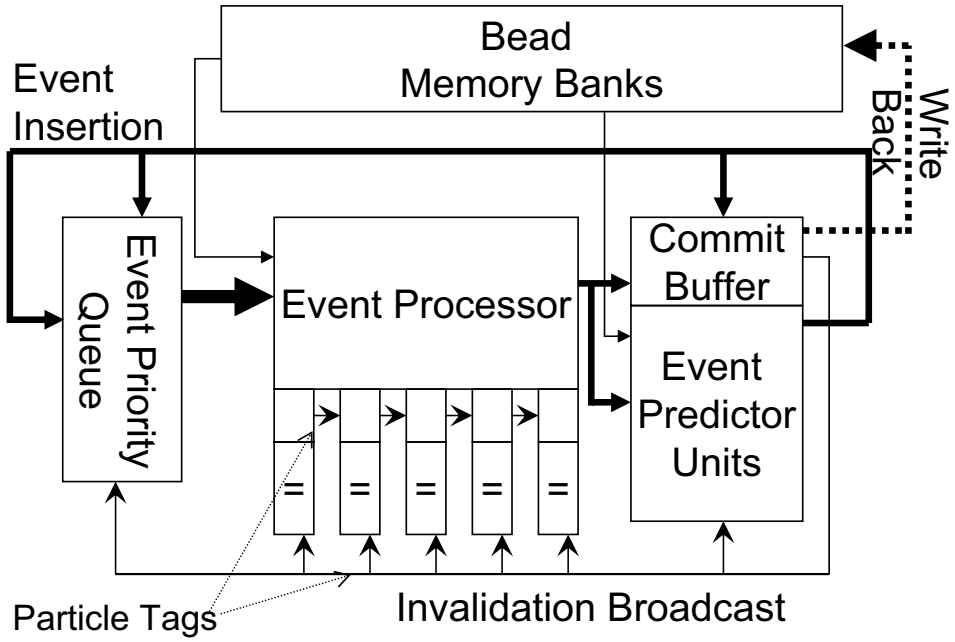

Figure 3·1: Block Diagram of DMD Simulator

As seen in Figure 3·1, 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 2.3. 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).

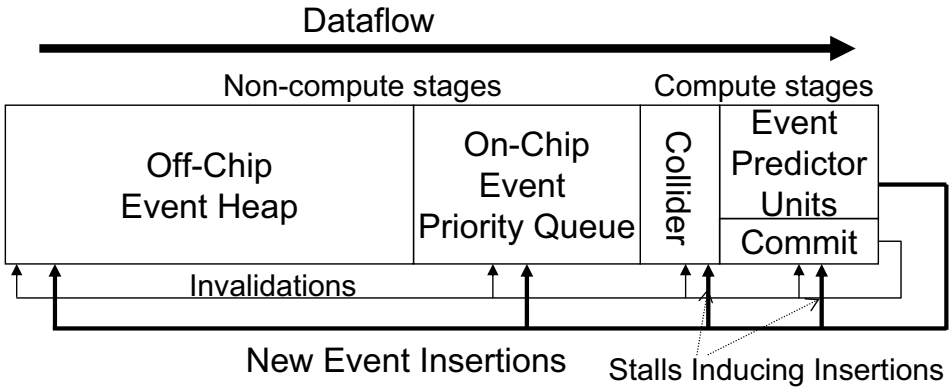

Figure 3·2: Conceptual Diagram of DMD Simulator

The overall concept of our design is shown in Figure 3·2. The entire system is effectively a time-keyed priority queue. At the front of the queue are the event processor (labelled "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 re-inserted into the queue at the appropriatelocation, based on the time tag. Insertion of events into the processing pipeline section of the queue requires special handling in order not to foul the computation in progress. However, insertion into the remaining portion of the queue, which makes up the overwhelming part of the system, is easily done. Reordering events here has no effect and the time order is preserved at the output.

The bead memory banks store the system state. Memory is interleaved in such a way that the contents of an entire cell neighborhood may be accessed at once (37).

Finally, committing an update 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: the next cell crossing (always) and the next collision (assuming it occurs before the cell-crossing), as in (19).

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. What happens when:

Case 1. Events need to be cancelled that are already in the event processors?

Case 2. Events need to be inserted into the event processors?

Case 3. An event entering the event prediction pipeline has potentially stale information because of an event ahead which 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 .5%). Moreover the effect of an unfilled hole is local: it only means that no payload was 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 at least 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 44%.

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 (26). 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., (3)).

Chapter 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.

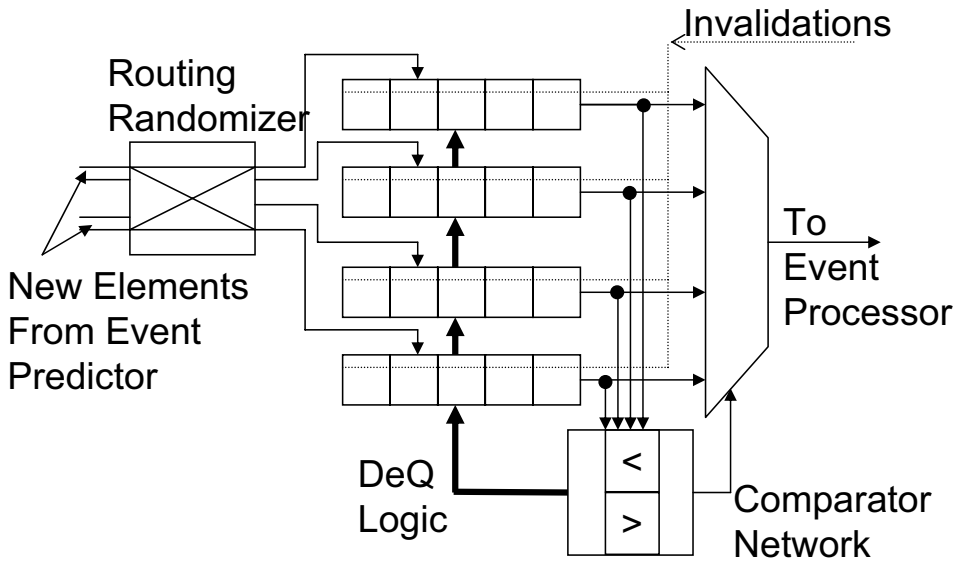

Figure 4·1: Four Insertion Event Priority Queue

4.1 Event Priority Queue

The event priority queue ensures that the event processor's inputs arrive in time order. This is accomplished by taking advantage of the unique broadcast mechanisms available in an FPGA. The queue is composed of four single-insertion shift register units, as seen in Figure 4·1. The event predictor presents two to four new elements to the routing network at each cycle. One of the twenty four possible routing scenarios is

pseudorandomly selected and each of the four shift register units determine the correct location to enqueue its new element. Simultaneously, the dequeueing logic is generated by examining the heads of each of the four shift register units, and choosing the next event.

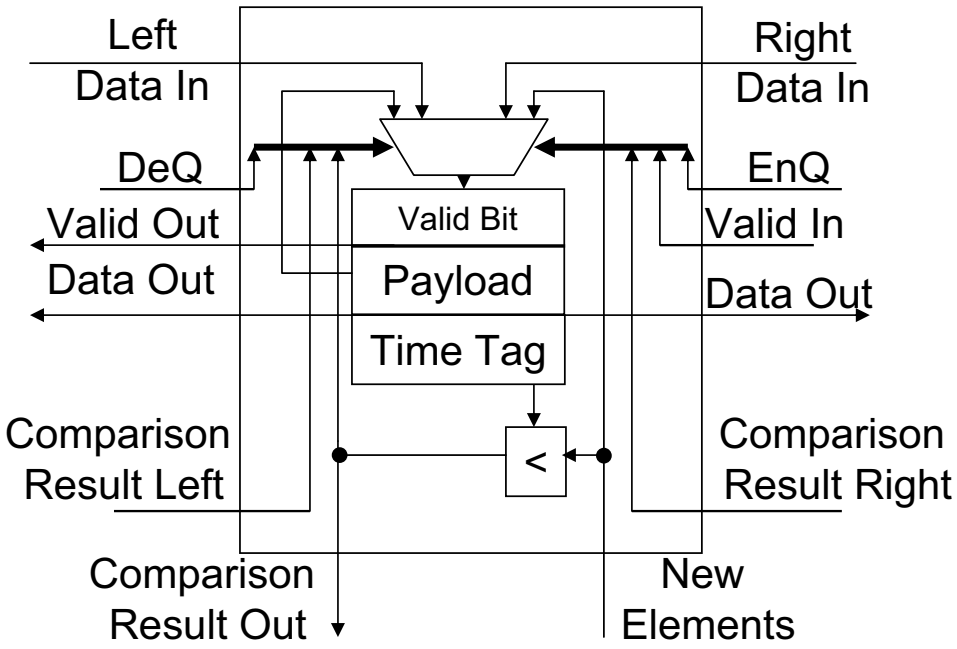

Figure 4·2: Single Insertion, "Scrunching" Priority Queue Unit Cell

The shift register units themselves are an extension of the hardware priority queue described in (25). Each shift register unit cell contains a time tag, the payload (bead references), a valid bit, comparators and shift control logic (see Figure 4·2). 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 enqueing, the next neighbor is examined. When only enqueing, the previous neighbor is chosen. There are four actions which the shift register control logic can indicate:

Action: Shift Forward

Condition: When both the current and next cells time tags are lesser than the new

element's.

Action: Stay Put

Condition: When both the current and next time tags are greater than the new element's, or the current and previous tags are both lesser.

Action: Shift Backward:

Condition: When the current and previous time tags than are greater than the new element's.

Action: Insert New Element

Condition: When the current time tag is greater and the next time tag is lesser, or the current is lesser and the previous greater.

With up to four enqueue operations per clock cycle and only a single dequeue, our event priority queue would quickly fill. However, in addition to the insertions, invalidations are broadcast to each element of the queue every clock cycle. 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 (scrunching). 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 will not overwhelm the queue, and invalid "holes" are pushed to the back of the queue, yielding payload at the queue head with a probability of about 60%, according to our hardware simulation.

4.2 Event Processor

The event processor handles cell crossing and collision update processing, computing them in as few cycles as possible to avoid stalls. The velocity and position update is the output. The implementation of the event processor is a straightforward pipelining of

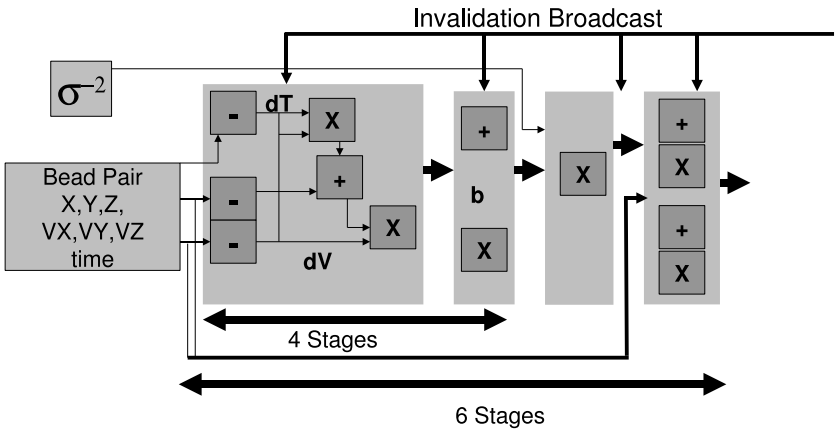

Figure 4·3: Event Processor Block Diagram

equation 2.5. As currently implemented, only hard-sphere and bonded type interactions are supported. In Figure 4·3 this is represented by the $sigma^{-2}$ input to the unit. The only subtlety here is that the division is replaced by a constant multplication, as all interactions take place at a pre-defined 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 collision or crossing. There need only be a single event processor.

4.3 Event Predictor

The event predictor generates the new events to replace those which 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 equation 2.4. 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

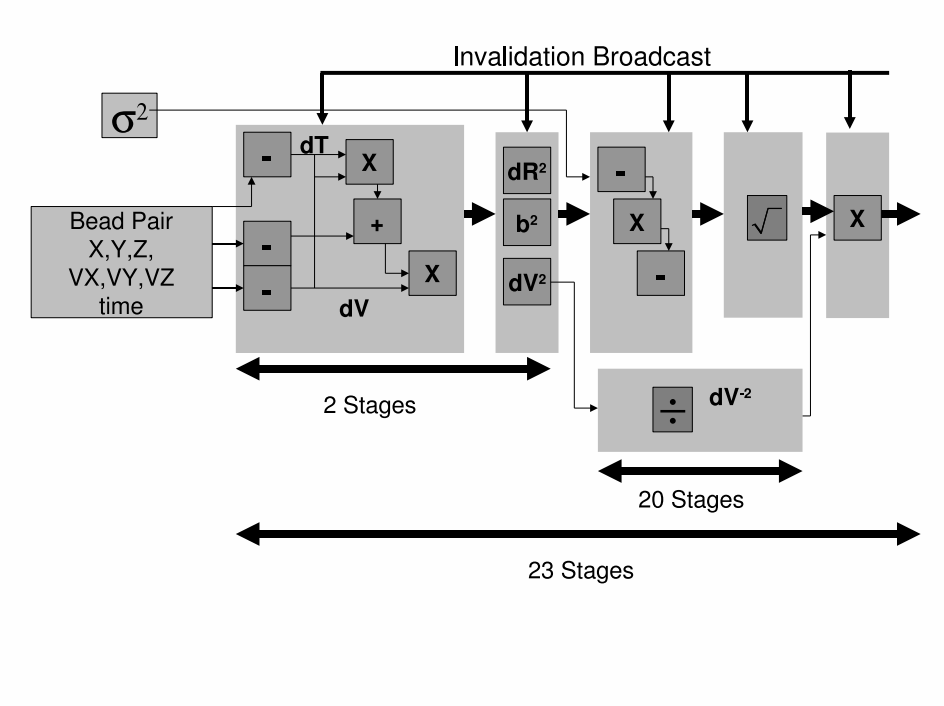

Figure 4·4: Event Predictor Block Diagram

the event processor, a set of shadow registers maintains state in the event of a stall. A number of event predictor units will need to be available, depending on the simulation density.

4.4 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 in proportion to the time-step size, during interactions. Second, it is only essential to preserve causal ordering 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 Angstrom simulation box, that cor-

responds to a resolution of 2.9×10^{-8} Angstroms in position. Full 32-bit precision is used in the event processor in order to minimize velocity variation. However, in the event predictors, the parameters used for covalent bond length are accurate only to 10^{-2} Angstroms. This corresponds to twelve bits, allowing us to reduce the precision used for the inputs. The output precision is returned to 32-bits over the course of its operation.

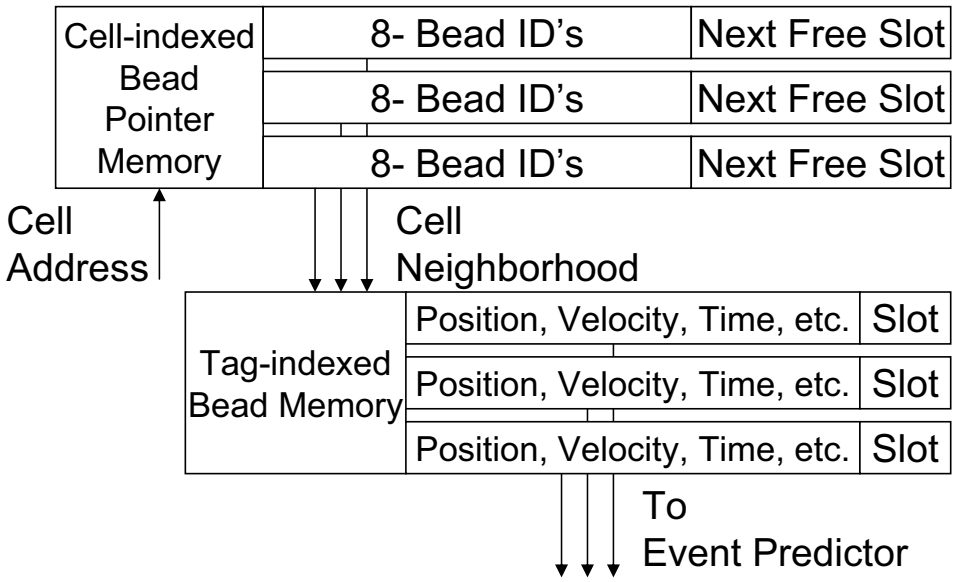

Figure 4·5: Bead Memory Block Diagram

4.5 Bead Memory

There is one type of read operation nd 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 4·5. The bead pointer memory is interleaved by position and contains eight pointers to beads in the addressed cell. The cell size is equal to, or just larger than the bead diameter, and as such, at most eight beads can be physically contained in a given cell. Our hardware simulation indicates indicate that four pointers per cell may be adequate for hard-spheres, though this is bound to vary with specific simulation paramters. 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. Since beads sharing a neighborhood will induce coherence stalls upon entering the commit buffer, we are safe from hazards. 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.

Chapter 5

Performance and Accuracy

5.1 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 tools. Arithmetic retains 32-bits of precision through binary scaling except where reduced precision is safe (see Section 4).

5.2 Validation

As with traditional MD (14), validation requires several 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 (29). The effect of reduced precision must be measured indirectly, e.g., by measuring energy fluctuation. Unlike MD, DMD is energy conserving to the precision of the arithmetic. As can be seen in the snapshot shown in Figure 5·1, 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. In any case, precision will remain a design parameter to be varied by the computational biologist as a part of the DMD interactive

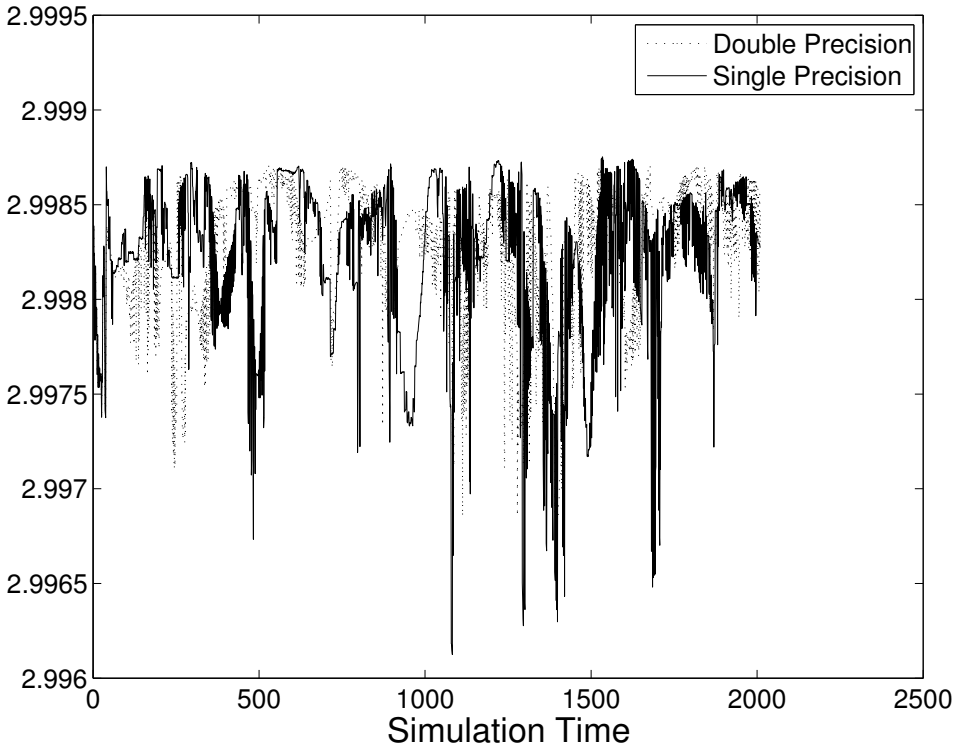

Figure 5·1: Total Energy vs. Cycle for Single and Double Precision Aimulations

experimental protocol.

In addition to the modified Rapaport code, we have developed a C-based simulation of the hardware described above, for hard spheres. This code is cycle-driven, and is faithful to the hardware with the notable exception that it currently uses floating point arithmetic. Still, it is a useful tool for experimenting with the dynamics of the simulation and exploring hardware parameters' affect on the simulation. Using this code, we have produced several validations of our assumptions.

Key to maintaining a high event throughput is the relatively uniform distribution of insertion of new events into the Priority Queue. Experiments have shown that this is the case for the modified Rapaport code, however, our choice of Lubachevsky's "Lazy" insertion mechanism, which only inserts the one or two next events for each bead, is significantly different than that of Rapaport. However, as can be seen in Figure 5·2,

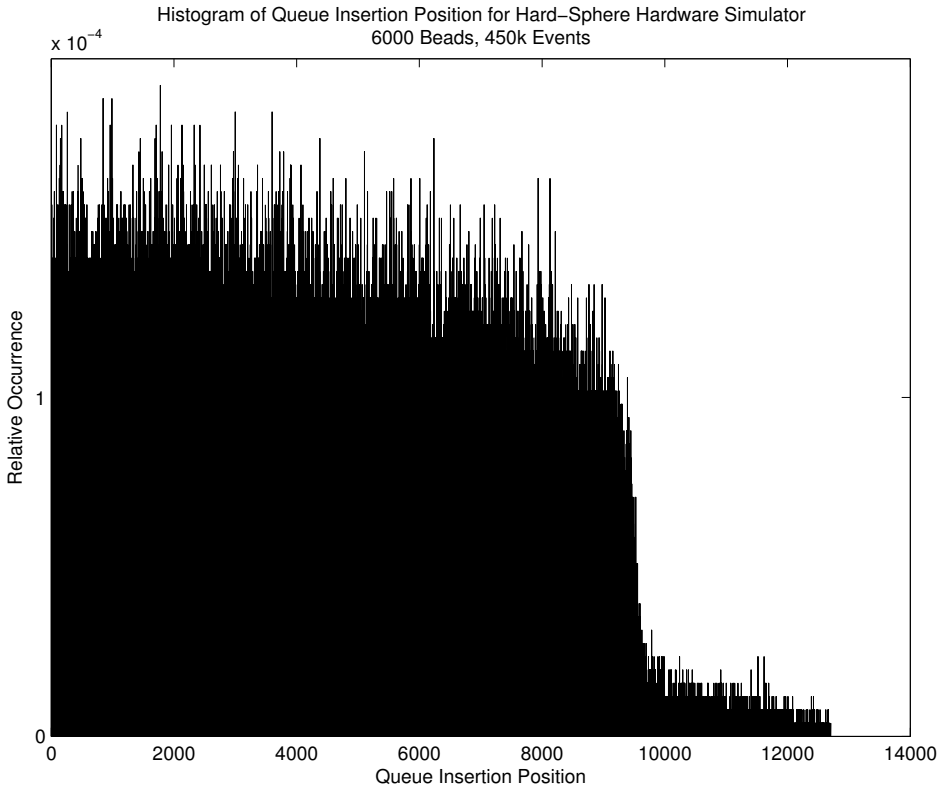

Figure 5·2: Histogram of Queue Insertion Position

the near uniform distribution of queue insertions is maintained.

The radial distribution function is another measure of simulation accuracy, which describes the organization of beads around any given bead (29). For a simulation of hard spheres with periodic boundary conditions, there will be a discontinuity at the hard sphere radius, and a uniform distribution beyond that. As can be seen in Figure 5·3 both the modified Rapaport, and the hardware simulator yield similar results.

5.3 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

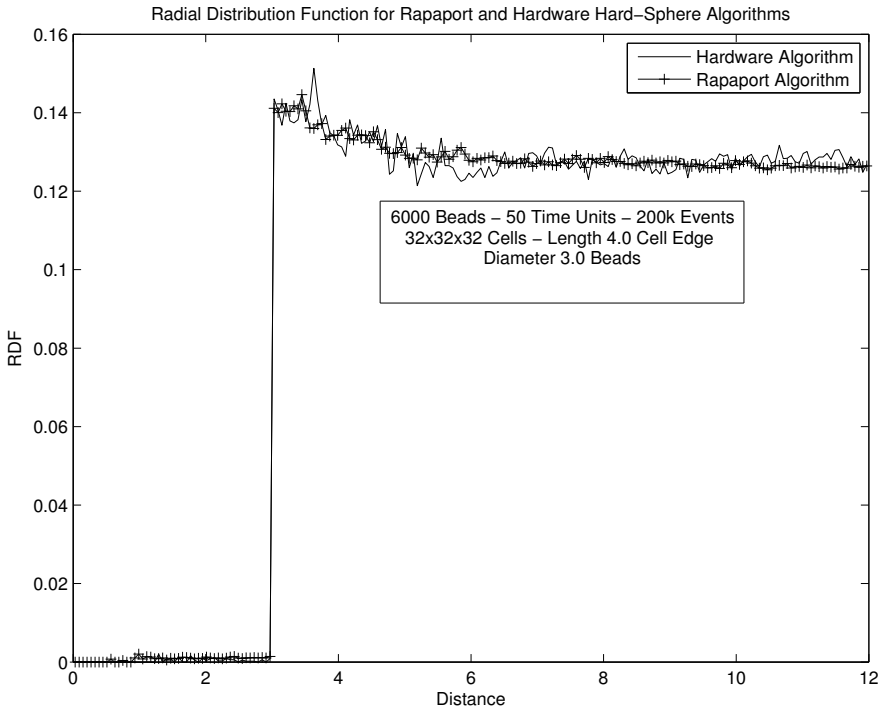

Figure 5·3: Radial Distribution Function for Rapaport and Hardware simulator Hard-Spheres

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. The critical path is currently the number of cycles it takes to insert predicted events into the priority queue. As described in Section 4, this takes 2 cycles in our current implementation. The resulting performance for an ideal execution (no stalls) is one event per two clock cycles.

3. Stalls per clock cycle. This was discussed in Section 3.2. According to our hardware simulator, the number of stalls per cycle in the current design is approximately .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 implemented on the Xilinx Virtex-II-Pro XC2VP70 -5, which we use on our Annapolis Microsystems Wildstar II-Pro, 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 about 150 on-chip stages. When moving to the Xilinx V4LX200, the size of the processing section remains similar, resulting over 1000 stages fitting on-chip. The Event Processor, Event Predictor and Priority Queue have been synthesized using Synplify Pro 8.0, and placed and routed using the Xilinx toolflow. Timing was successfully constrained to a 10 ns clock cycle time. Newer FPGA architectures and simulation specific optimizations should allow more stages to fit 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.

Serial Performance

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

(29), modified by us to handle covalent bonds, and from Donev, developed for (10). 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$\times$ for the smaller models.

Chapter 6

Conclusion

6.1 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 duplicate 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 (5; 15; 23; 35). As stated earlier, 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 comptutational models, leading to the necessity of flexible DMD simulator design.

There is much work yet to be done. In particular, the off-chip priority queue and its controller, and the bead memory controller needs to be implemented. In addition. we hope to work with biologists to realize more complex force models, and address the biologcially significant molecules currently being examined by software.

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] Alder, B.J., Wainwright,T.E. Studies in molecular dynamics. I. General method. *Journal of Chemical Physics 31*, 2 (1959) 459466.

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

[4] Azizi, N., Kuon, I., Egier, A., Darabiha, A., and Chow, P. Reconfigurable molecular dynamics simulator. In *FCCM* (2004), pp. 197–206.

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

[6] Chandy,K., Misra, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering, SE-5:5*, (1979), 440-552.

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

[8] Dokholyan, N., Buldyrev, S., Stanley, H.E., Shaknovich, E. Discrete molecular dynamics studies of the folding of a protein-like model. *Folding and Design 3*, (1998), 577587.

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

[10] 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.

[11] Erpenbeck, J., Wood, W. Molecular dynamics techniques for hard-core systems. in *Statistical Mechanics* B. J. Berne, ed. Volume 6 of *Modern Theoretical Chemistry*, Plenum Press, New York. (1977), Ch.1, 1-40.

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

[13] Gu, Y., and Herbordt, M. C. FPGA-based multigrid computations for molecular dynamics simulations. In *FCCM* (2007).

[14] 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.

[15] Keane, J., Bradley, C., and Ebeling, C. A compiled accelerator for biological cell signaling simulations. In *FPGA* (2004).

[16] Kindratenko, V., and Pointer, D. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *FCCM* (2006).

[17] 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. *J. Comp. Chem. 18*, 12 (1997), 1546–1563.

[18] Lubachevsky, B. Bounded lag distributed discrete event simulation *Proceedings of the 1988 SCS Multiconference, Simulation Series, SCS 19*, 3. 183-191.

[19] Lubachevsky, B. Simulating billiards: Serially and in parallel. *International Journal of Computers in Simulation 2* (1992), 373–411.

[20] MacKenzie, Tropper Parallel Simulation of Billiard Balls using Shared Variables. *10th Workshop on Parallel and Distributed Simulation (PADS '96)*, 1996

[21] Marin,M., Risso, D., Cordero P. Efficient Algorithms for the Many-Body Hard Particle Molecular Dynamics *Journal of Computational Physics 109*, (1993), 306-317.

[22] McCammon, J.,Gelin, B.,Karplus, M. Dynamics of folded proteins *Nature 267*, 585 - 590 (1977).

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

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

[25] 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.

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

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

[28] Rapaport, D. The Event Scheduling Problem in Molecular Dynamic Simulation. *Journal of Computational Physics 34*, (1980), 184-201.

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

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

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

[32] Sigurgeirsson,H., Stuart, A., Wanz, W. Algorithms for Particle-Field Simulations with Collisions *Journal of Computational Physics 172*, (2001), 766-807.

[33] Smith, S., Hall, C.K.,Freeman, B.D. Molecular Dynamics for Polymeric Fluids Using Discontinuous Potentials *Journal of Computational Physics 134*, (1997), 1630.

[34] 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.

[35] Tripp, J., Mortveit, H., Hansson, A., and Gokhale, M. Metropolitan road traffic simulation on FPGAs. In *FCCM* (2005).

[36] 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.

[37] VanCourt, T., and Herbordt, M. Application-dependent memory interleaving enables high performance in FPGA-based grid computations. In *FPL* (2006), pp. 395–401.

[38] Zhuo, Y., Karplus, M., Wichert, J., Hall, C. Equilibrium thermodynamics of homopolymers and clusters