A COMPREHENSIVE INTEGRATION AND ANALYSIS OF DYNAMIC LOAD BALANCING ARCHITECTURES WITHIN MOLECULAR DYNAMICS

by

Christopher Reed Rogers

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Dr. Brandon Eames Major Professor	Dr. Aravind Dasu Committee Member
Dr. Wei-Ren Committee Member	Dr. Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2009

Abstract

A Comprehensive Integration and Analysis of Dynamic Load Balancing Architectures Within Molecular Dynamics

by

Christopher Reed Rogers, Master of Science

Utah State University, 2009

Major Professor: Dr. Brandon Eames
Department: Electrical and Computer Engineering

The world of nano-science is an ever-changing field. Molecular Dynamics (MD) is a computational suite of tools that is useful for analyzing and predicting behaviors of substances on the molecular level. The nature of MD is such that only a few types of computations are repeated thousands or sometimes millions of times over. Even a small increase speedup or efficiency of an MD simulator can compound itself over the life of the simulation and have a positive and observable effect. This thesis is the end result of an attempted speedup of the MD problem. Two types of MD architectures are developed: a dynamic architecture that is able to change along with the computational demands of the system, and a static architecture that is configured in terms of processing elements to be best suited to a variety of computational demands. The efficiency, throughput, area, and speed of the dynamic and static architectures are presented, highlighting the improvement that the dynamic architecture presents in its ability to provide load balancing.‘

(77 pages)

For Chrissa and Liam.

Acknowledgments

It would be remiss of me to pass on the opportunity to thank and acknowledge Dr. Brandon Eames for his faith in me. I appreciate his inspiration and desire to make me the student he knew I could be.

I would also like to acknowledge my wife's never-ending patience and support as I convinced her against her will to stay in school just a little longer.

Chris Rogers

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Molecular Dynamics Introduction

Molecular Dynamics (MD) commonly refers to a set of algorithms or packages designed to model or simulate dynamic particle interaction on the molecular or atomic level. In essence, knowledge of how a substance behaves on the molecular level aids in predicting how it will behave on the macro level. Knowledge gained from studying such interactions has led to a myriad of scientific advances across several fields, such as protein folding, pharmaceutical research, and material sciences. For material sciences, Molecular Dynamics simulations have yielded vast amounts of knowledge concerning fracture control. This subset of material science research examines how different substances handle stress as a function of pressure, force, and temperature. MD can aid in the development and innovation of new materials, as well as improve upon existing methods of material strengthening [1]. Protein folding is the process by which a string of polypeptides takes a three-dimensional shape. As the hydrophilic and hydrophobic amino acids take their place in the final state, there are steps that are taken which result in an unnatural final position. The process these steps take are still unknown to the scientific community. Scientists and geneticists are led to believe that many neurodegenerative disorders have their cause in these unnatural final states. As scientists understand more about the cause of a disease the easier treatment developments can occur [2]. Molecular Dynamics provides an ideal method by which research can continue to move forward in these fields.

1.1.1 Molecular Dynamics Simulation

MD simulations start with known mass, initial position, and initial velocity of each particle in a given substance or system. For example, a microscopic amount of pure water

will contain a certain number of H_2O molecules. All of the molecules considered together comprise the system. With the knowledge of the location of each molecule, it is possible to calculate the distance between all pairs of molecules in the system. The degree of separation between two molecules contributes to a force between the pair. In some cases, the force between the two particles attracts while in others it causes mutual repulsion. Two particles considered for the assessment of force are sometimes referred to as *neighbors*. If the distance between neighbors is sufficiently large, no noticeable force between them is observed. The net force accumulated on a particle by all other particles within its vicinity result in changes to the particle's position and velocity. MD simulation seeks to determine the impact of these cumulative forces on the particles of a system over several successive time steps. The simulation calculates inter-particle distances, net cumulative forces, and position and velocity updates for each particle in the system. Once the force, new position, and new velocity for each particle in the system is calculated for a given time instant, these results are folded back into the simulation and the process begins again for a new time step. The time-step simulation is typically executed for a fixed number of steps, or until some desired point in a simulated reaction is observed.

Depending on the objective of the MD simulation, the number of atoms can vary widely. Proving a specific concept, or zooming in on a small part of a substance to view the effect of its simulation only necessitates a few thousand atoms. Consequently, the computational load of an MD simulation ranges widely from project to project. Nanotechnology-oriented research projects are known to comprise a range from 3000 to 6000 atoms [3]. For large-scale MD projects, the demands are drastically increased. For example, to research protein folding, IBMs Blue-Gene/L super computer has been developed with the goal in mind to perform massively computationally intensive applications. A dedicated MD code was developed and ran on the Blue-Gene/L platform. The simulation sustained a speed of 100 trillion floating-point operations per second, which is a tera-FLOPS or TFLOPS, throughout the operation. MD simulations of this nature push the limit of the supercomputer's abilities. These simulations can take millions of atoms [4].

As has been shown, MD simulations can be used to research and prove many useful concepts. Scientists and engineers have often collaborated on taking MD simulations to more intense platforms in order to fully explore the areas MD research has to offer. The floating-point operations that MD requires can be very expensive in terms of computation power and time. The average computer cannot provide the brute force means of simulating the effects of millions of atoms. This sort of problem lends itself to exploring MD simulations through parallel computing, and using supercomputers when available. Sometimes, as in the case of Blue-Gene/L, the parallel computing is the means by which the end of super computing is met. There are scalability issues that arise in parallel computing projects, and developers have been examining reconfigurable hardware as a platform for accelerating MD computations. By developing a customized architecture that is capable of performing MD simulations, a highly efficient implementation of the same can be developed and used. In a simulation where thousands of computations take place, the smallest of increases in efficiency will be repeated many times over and translate to a large effect on the design.

1.1.2 Load Balancing

Not only can the computational requirements of an MD simulation change from one project to the next, they change within a single simulation one particle to the next. Depending on the type of substance and environmental conditions being simulated, a particle may interact with several others at a given time instant, while other particles may interact with relatively few particles. Further, as the simulation proceeds, the spatial arrangement of particles varies according to the dynamics of the system, leading to a high degree of unpredictability and variance in the types and frequency of computations required by the simulation. This unpredictability and variance can be characterized as a load imbalance between the force and the distance calculations. Because there are different time demands and latencies between the calculations of force and distance, it is difficult to predict beforehand how much time is necessary to spend on distance or force calculation. When considering the variation in load requirements imposed by changes in the environmental parameters and simulation substances, a one-size-fits-all static hardware allocation presents a challenge

when the designer attempts to maximize computational efficiency.

The difficulty of predicting the computation load makes field programmable gate arrays (FPGA) appear a poor choice as a platform in developing an MD architecture. However, FPGAs offer the ability to make an architecture fit the problem. In addition, all processing elements on an FPGA are capable of spatial and temporal parallelism. This means that it is possible to create several smaller independent processing elements that can function in parallel. Despite the advantages offered by an FPGA platform, the problem with load balancing still looms. How much of the FPGA's resources are to be allocated to force, distance, or velocity and position updates comprise the exact problem of load balancing.

In an ongoing research project, the authors and others have developed an approach to dynamic load balancing targeting an FPGA. The approach focuses on the use of a Flex processor. The Flex processor is an MD specific pre-designed FPGA-based processing element that can switch between distance calculation and force calculation, the two most common computations in the MD simulation. This alleviates the imbalance in data computations by allowing for processing elements to change modes and address computational load imbalances on the fly. Hence, for a particle with many neighbors whose proximity is such that a significant force results, the majority of which are too distant to exhibit a mutual force, a series of Flex processors can primarily function in distance calculation mode to quickly find those pairs of particles which are sufficiently close to warrant force calculation. When a sufficient number of such pairs has been obtained, the mode of Flex processors can be changed on demand to perform force calculations as needed. Thus in this case, most computational resources are dedicated to distance computation. In another scenario, consisting of a higher concentration of particles, more Flex processors could be set to force-calculation mode in order to balance the on chip resources dedicated to distance computation and force calculation. Further, the ability to change Flex processor modes on the fly allows the system to dynamically adapt to unforeseen changes in particle patterns that arise due to interaction dynamics.

We also approach dynamic load balancing by implementing a series of first in first out

(FIFO) buffers that feed particle data to the distance calculation units, and another series which store data between the distance and force calculation stages. Due to the fact that each computation stage is made up of multiple computation elements, the presence of FIFOs which can support reads and writes from multiple components has the effect of balancing load variations which arise at different locations across the same time step in the simulation. Further, the FIFO buffers decouple the design, making it more modular, and allows the most computationally intensive stages of the design to operate separately from each other and from the remaining components in the system. The use of Flex processors, together with decoupling FIFOs, facilitate load balancing decisions to be made at runtime, as the simulation proceeds. The system controller is free to allocate the low-level computational resources of the FPGA to where they are currently needed most.

1.1.3 Thesis Overview

By exploiting the pipeline and parallel computing capabilities of an FPGA, there is potential to create a highly efficient customized architecture that accelerates an MD simulation through addressing the issue dynamic load balancing. The incorporation and efficient use of the Flex unit is able to dynamically meet the computational demands of the MD problem and accelerate the simulation. However, lacking from the existing research is a quantitative evaluation of the impact of dynamic load balancing on an MD simulation. The existing research focuses on the development of an architectural approach for supporting dynamic load balancing on an FPGA. In this thesis both the integration and realization of dynamic load balancing in hardware, as well as the quantification of its performance against similar, but statically load balanced hardware architectures, are addressed.

It is not sufficient to state that dynamic load balancing is enough to accelerate MD simulations. This concept must be proven. As such, four total architectures have been created. The first of these architectures was developed by the author and two other researchers and relies on the Flex processor as the crux of dynamic load balancing. The remaining three were developed to be static implementations of an MD architecture simulator each with different numbers of processing elements in each computational stage in an attempt to address

load imbalances.

The subsequent sections of the paper describe the rationale behind and the design of our load balancing architectures for MD simulation. Chapter 2 discusses related work, Chapter 3 discusses the development of the custom dynamic load balancing architecture including all design concepts. Chapter 4 discusses the results and analysis of the custom design. Chapter 5 reviews the implementation, testing, and results of the competing static architectures in comparison to the Flex based architecture. Chapter 6 closes with observations and conclusions as well as a discussion of future work envisioned and the direction of the project. The bibliography is found at the end of all these chapters.

Chapter 2

Related Works

This section will examine and document the recent work and projects in reconfigurable computing as they relate to the concepts explored in our design. Topics of work discussed are:

1. Acceleration of MD simulations,
2. Scientific computing and MD architectures for FPGAs,
3. Load balancing in MD and computationally intensive FPGA applications,
4. MD domain specific languages,
5. Numerical computing on FPGAs,
6. Scalable MD architectures,
7. Fault tolerant FPGA architectures and automated HW/SW generation for FPGA targets.

All of these areas of reconfigurable computing offer insight into how to approach a problem such as Molecular Dynamics when the target platform is an FPGA. Our design differs in through our approach to dynamic load balancing. Our controllers and FLEX processing elements are capable of tailoring the behavior of the architecture to best meet the needs of the simulation. Each of the above mentioned topics will be explored in some detail for the rest of this section.

2.1 Acceleration of Molecular Dynamics Simulations

Several recent research efforts have focused on attempting to speed up MD simulations. For example, an FPGA has been used as a floating-point coprocessor for acceleration of N-body problems (such as MD) in a work enacted by Lienhart et al. [5]. Fully pipelined, complex arithmetic units using floating-point adders, multipliers, dividers, and square root units were created from efficient low-level primitives. A mixture of 60 of these units was placed on a Xilinx Virtex2 FPGA. Sustained throughput of up to 3.9 giga-FLOPS (GFLOPS) was attained under ideal conditions. Cordova and Buell [6] developed a novel scaling technique to accelerate MD simulations in a reconfigurable setup, particularly for a super computer. The entire MD design is done in floating-point arithmetic. The authors made use of a multi-adaptive processor, in which M data paths and a data access mechanism independent of the MD kernel reduced computation costs down to O(N/M). The entire design was ported to the SRC-6 supercomputer with promising results. Specifically studied by Wolinski et al. [7] is the effect of using the smooth particle mesh Ewald technique, which is a modification of the standard Ewald method, in hopes of accelerating the MD simulation. The Ewald summation method computes the interaction energies of periodic systems, particularly the electrostatic energies. This approach has an advantage in rapid convergence. Using this method, a speedup of 2.7x to 2.9x was realized over the software design.

2.2 Molecular Dynamics and Scientific Computing Architectures for FPGAs

With the emerging low-cost FPGA alternative, it is not surprising that many have attempted to port MD architectures to an FPGA. A recent approach to solving MD problems involves the use of Application Specific Processors (ASPs) on FPGAs [8]. Unique computational units are derived, such as pair generators, Lennard-Jones potential calculators, and acceleration update blocks. A mixture of these blocks is placed upon the FPGA. Different data sets will yield different performance levels with different architectures. Difficulties arise in determining the ideal FPGA configuration for a specific data set. A research effort to map the Lennard-Jones and Coulombic force computations to an FPGA [9] yielded nominal speedups. Rather than using supercomputers or custom hardware, this research involves

solely commercial-off-the-shelf (COTS) components. An investigation was performed on the tradeoff between accuracy and speed, by adjusting the bit-precision of floating-point numbers. Depending on the accuracy required, this system achieves speedups between 31 and 88 times over a conventional PC approach. A highly optimized double precision floating-point based deeply pipelined architecture that can offer 3.9 GFLOPs of performance was explored by Scrofano and Prasanna [10]. Wolinski et al. made other efforts in the area of putting Molecular Dynamics on a reconfigurable machine in order to take advantage of spatial and temporal parallelism [11]. The MD design was split up into hardware and software by Trouw et al. [12] in order to derive a neighbor list in software and then fed to hardware, where the data was managed by two controllers, a read and write controller. These controllers fed the data in turn to the force calculators. When all the force had been calculated for a neighbor list the force information was returned to software to be held until the appropriate time for position and velocity update. A variety of different hardware designs, including a write-back design, were explored. Fluid dynamics through MD [13] was explored using a two-dimensional systolic array of cells, where each cell had a multiplication and accumulation unit that would be used to implement a computational analysis of the substance being simulated. The entire design was put on an Altera Stratix II FPGA. All operations were done in floating-point arithmetic, and a speedup of 7x over a Pentium4 implementation of the same was realized. This further illustrates the ability of an FPGA to perform complex calculations regardless of slow clock times and still have the ability to perform with lower power and area. VanCourt et al. [14] also developed an FPGA implementation without any special techniques. The intent of the research was to show the power of the FPGA platform for N-body problems such as MD. The motion estimation in this design was Verlet, as was used in GROMACS, an independently developed MD software suite [15], and in our design. A 57x speedup of the same design in software was attained. The authors are hopeful for a speedup of double that number with future work.

2.3 Load Balancing in Molecular Dynamics and FPGA Computing

Other efforts than our own in accelerating MD simulations through dynamic load

balancing were explored as well. MD simulations using parallel computers [16] used the concept of permanent cells within a dynamic load balancing method in an effort to curb inter-processor communication overhead. The simulated block of particles or molecules was divided up in such a way that a processing element was given a geometrical slice. This was referred to as domain decomposition. Dynamic load balancing is achieved when these geometrical slices are moved around to different processing elements. However, not all cells are moved and this is where the idea of permanent cells is introduced. The idea of permanent cells was also used [17] to monitor the theoretical effective ranges of dynamic load balancing to see where it is more and less effective. The study showed that dynamic load balancing for their platform is more effective when a smaller number of processing elements is used. The simulation was again performed using parallel computers, in this case, the T3E. Load balancing may also be achieved via generalized dimension exchange (GDE) [18]. GDE is realized by four parts: load measurement, information exchange, initiation, and load balancing operation. The analysis of the study focused on keeping all processing elements busy by testing the design with a variety of different parameters. The cost in overhead of tracking a processing element's work can negate the advantages that sorting the data can bring. Running the MD simulation with load balancing achieved through GDE resulted in a 26% speedup over the simulation where no load balancing was used. The divide-and-conquer method of load balancing is also useful [19]. The authors developed an MD simulator that is extremely physically and chemically accurate; capable of simulating many kinds of atoms, molecules, particles of all kinds of structure-lattice, fluid, gas, and solid. The total volume of the system is divided into P subsystems of equal volume, and each subsystem is assigned to a different node in an array of P processors. The load balancing aspect is achieved by separating the system not in physical space but in computational space through curvilinear coordinate transformations. FPGA resource managers are employed [20] to dynamically change the needs of computing on an FPGA. The resource requirements are made known to the resource manager in the form of directed flow graphs and the resource manager, which resides in software, makes the necessary changes to the FPGA. The intent of the

authors is to demonstrate conclusively that FPGA computing is very competitive in terms of cost and throughput with traditional methods of computing and some applications can in fact be sped up through the concurrent models that FPGA computing supports. Dynamic load balancing is also realized through grid computing [21]. The authors test their technique of dynamic load balancing by first identifying which processors in a given network are neighbors, that is, which processors have immediate communication between them. The algorithm that they test, a PDE that describes chemical computations, is an asynchronous iterative algorithm. In a previous work, [22], the authors showed that load balancing was already a good way of reducing computation time of such algorithms. The results show that for certain conditions, such as setting load balancing requirements for all iterations of the process rather than for one iteration, dynamic load balancing greatly improved computation time of asynchronous iterative algorithms.

2.4 Molecular Dynamics Domain Specific Languages

MD simulations have also been the subject of acceleration through the development of MD specific languages. MD architectures have been developed using XML and Molecular Dynamics Language (MoDL) to implement MD visualizations [23]. The goal in mind was to simulate MD interactions through VRML plug-ins available for any common web browser. Hurdles included developing a tool that enabled chemists and physicists without a strong computer background to develop the MoDL implementation to meet the criteria of the simulation. The MoDL language itself was also developed to be purposefully simple and straightforward. The bulk of the work done was in the development of this tool, which not only aided in the initial setup of the MD simulation but also in converting the MoDL language to a plottable VRML implementation. All this work was done using Perl and Tk. Combining FPGA implementation and domain specific languages [24] introduces a novel interpolation strategy and a semi-floating-point design to speedup MD simulations over the PC. By using the resources native to FPGAs, most notably look up tables, the authors developed an interpolation technique that approximates a very accurate force calculation in order to attain speedup over conventional force derivations. Domain specific languages

are also used to more efficiently make use of available hardware for MD simulations [23]. ProtoMol, a product of Notre Dame, is used to develop the specifics of the MD simulation for the FPGA part of the design to simulate. The FPGA and ProtoMol implementation yielded a speedup of 5.5x over a 2.8GHz Xeon PC, with acceptable tradeoff in accuracy. The multigrid approach to MD [25] investigates spatial ordering of the system through an architecture on an FPGA platform. The MD system is sectioned off into grids and processed iteratively in all neighborhoods. This method is similar to extrapolation between coarser and finer grids. The FPGA design is implemented with ProtoMol, and a speedup of 5x to 7x over a software design of the same is achieved.

2.5 Numerical Computing on FPGAs

As MD is composed of numerical computations to simulate particle movement, it was pertinent to explore the work that has been done in the field of scientific numerical computing. Monte Carlo methods have applicability to MD. The Monte Carlo method is a set of computational algorithms that simulate the behavior of a given physical system, in the case of this research project, anisotropically polarizable particles as affected by an external electrical field. The Monte Carlo method, while differing from MD in that the former is stochastic while the latter is deterministic, still exhibits the need for quick parallel computing. A dipole method of computation was introduced both on an FPGA and in software using the Monte Carlo method [26]. A speedup of 43x was realized. Choi and Park analyzed FPGA memory blocks and how using the redundant bit and word configurations in the embedded array blocks can aid in yield enhancement [27]. In operationally and computationally rich systems, such as multimedia targets, FPGA implementations can be slow. Increasing the yield on a target can increase the performance of the application. Several mathematical models to accomplish this goal were derived and implemented. A technique to compress the configuration bit-rate technique also increases MD performance on FPGAs [28]. This compression allows for less memory utilization for configuration states on any SRAM based FPGA. Results showed that the time required to configure the FPGA was not affected by the compression-decompression scheme. Memory savings up to 41% were

reached. Optimizing the number of functional units for MD architectures on an FPGA is a useful technique of improving MD performance when considering the needs of a diverse domain [29]. The point of the algorithms is to develop a balance of hardware load and processing elements that optimizes area utilization and throughput of a given design. Steiger et al. [30] developed a run-time system for FPGA based embedded systems, wherein a set of hard deadline real-time tasks are given and a guarantee based schedule is derived. This schedule is derived via two heuristics and then the two are applied to a one-dimensional and two-dimensional area models. Experimental results show that the two-dimensional model is more choice in that it does not depend on task aspect ratios.

2.6 Scalable Molecular Dynamics

By developing an MD architecture that is scalable, it is possible to increase overall throughput by simply augmenting the amount of processing power. G̈uided simulations,äs named by Yu. [31], attempt to make up for the time problems in MD simulations by scaling the design up to 1000 processors through data driven time parallelization approaches. Speedup is also achieved through prediction and verification of the MD process. Simulations were done of various make-ups and results were promising for future work. Scalability may also be realied through parallel decomposition methods and message passing techniques that reduce inter-processor communication overhead in an attempt to scale MD to high performance commodity cluster systems [32]. An MD code, named Desmond, was developed to incorporate these changes.

2.7 Fault Tolerant FPGA Architectures and Automated HW/SW Generation

Fault tolerance is not related to MD research, however it does contain techniques for numerical processing on FPGAs that may be constrained to aid in MD processing on the same platform. Fault tolerant logic blocks are checked for fault tolerance using roving self test areas, or STARs, to perform testing, diagnosis, and reconfiguration [33]. Their addition to the project is in the reusing of faulty logic blocks in order to increase available spares. Using the defective blocks increases the number of tolerable faults. VHDL

and RIF files that represent hardware and reconfigurability within the hardware may be automatically generated and be capable of using run-time configuration [34]. The design must be correct before place and route actions are taken. Power consumption may be reduced and system performance improved through the design's ability to be reconfigured. Shang and Jha developed a co-synthesis algorithm [35] in order to optimize system price and power consumption of multi-rate low power real-time distributed embedded systems composed of FPGAs. The co-synthesis algorithm is in the family of algorithms designed to automatically produce hardware-software architectures for distributed embedded systems. Coupled with a proposed dynamic priority multi-rate scheduling algorithm, experimental results show that schedule lengths are reduced on average by 34.3% and reconfigurable energy by the average of 40.4% when compared to the same distributed system without using the co-synthesis and scheduling algorithms.

Chapter 3

Molecular Dynamics Overview and Architecture

In this chapter the theory behind the architecture of Molecular Dynamics is presented. Each processing element is examined in detail. Where applicable, the physical science behind the architecture is also reviewed. This is followed by a presentation of the details of the dynamic load balancing architecture, including the Flex processor. The buffering, rate matching, and decoupling techniques as implemented on the FPGA, and how they contribute to dynamic load balancing, will also be examined.

3.1 Molecular Dynamics Algorithm

MD simulations rely on classical mechanics and are known as n-body problems. The basic process of calculating the distances between particles, deriving the force, and updating position and velocity is repeated many times over. There are three general steps to complete in order to have a properly functioning MD algorithm. These phases are distance calculation, Lennard-Jones potential calculation, and Verlet calculation. This is shown in fig. 3.1.

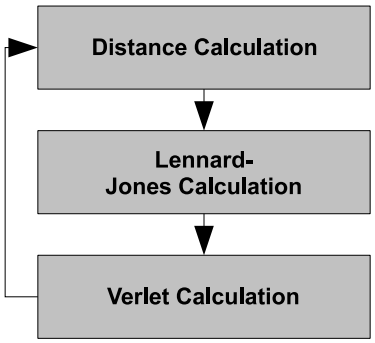

Fig. 3.1: Basic phases for an MD simulation.

The algorithm starts with a set of particles whose initial positions and velocities in a given system are known. Each particle in the system is examined in relation to every other particle. First, a Euclidean distance is computed between all particle pairs. For particle pairs which are deemed to be sufficiently close (i.e., within a pre-determined cutoff radius), the Lennard-Jones force is then computed. The net force on each particle is accumulated, and is used to update its position and velocity. This algorithm is given in algorithm 3.1.

Algorithm 3.1 Algorithm for an MD simulation.

```
For time step t:
For each particle
    For all other particles
        Compute distance
        If distance <cutoff radius
            Compute LJPC
            Accumulate net force on candidate particle
            Compute and store in temp buffer new velocity and position values
        Else
            Skip to next particle
Do Verlet update
```

It is worth noting that all particle pairs are compared; that is, the distance calculations are applied to all pairs of particles in the system. Only a subset of particle pairs are passed on to the Lennard-Jones calculation stage. This down selection introduces the primary source of load imbalance. The number of Lennard-Jones potential calculations that will be required during simulation is not known at any point prior to run-time, and further, can vary at run-time. In customized architectures which dedicate specific resources to individual classes of computations, this load imbalance can hinder throughput and speed of simulation.

3.2 Molecular Dynamics Architecture

In this section the hardware architecture realization of the MD simulation will be discussed, first from a system-level perspective, followed by a detailed description of each of the design components. These realizations are referred to as processing elements. Following the architecture discussion, we present the rationale behind the use of an FPGA as our target

platform, and what measures were taken to exploit specific hardware features of FPGAs in order to achieve high computational throughput. Figure 3.2 depicts a high level view of the hardware architecture employed in MD simulation. The following subsections will describe each of the major computational components involved in this design, namely Distance Calculation, the Inverter, Force Calculation, and Verlet Update. Each subsection introduces the mathematics driving processing element computation, followed by a discussion of the resulting hardware implementation. We will then transition to a discussion of the lack of load predictability of an MD simulation and show the approach of using the Flex processor and buffering techniques to dynamically compensate for load variance.

3.2.1 Feeder FIFO

The initial data as required by the MD architecture simulator is stored in a processing element called the Feeder FIFO. It has a controller that monitors the state of the simulation and controls the data that is processed by the distance calculation (DC) units. This data is stored in two sets of FIFO buffers. One set of buffers contain all of the data required by the distance calculation stage. The other set is identical in size and collects the updated position data from the Verlet unit as they are calculated for each candidate particle. Once a time step is complete, and all particles have been examined as candidate particles, the Feeder FIFO controller overwrites the first set of buffers with the new position data and a new time step can begin. A chart of this is shown in fig. 3.3.

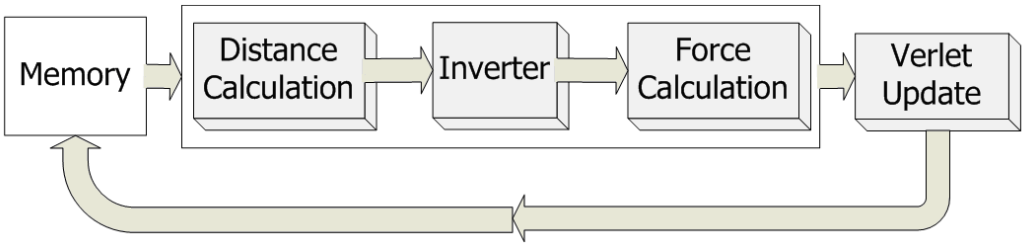

Fig. 3.2: High-level view for our MD architecture.

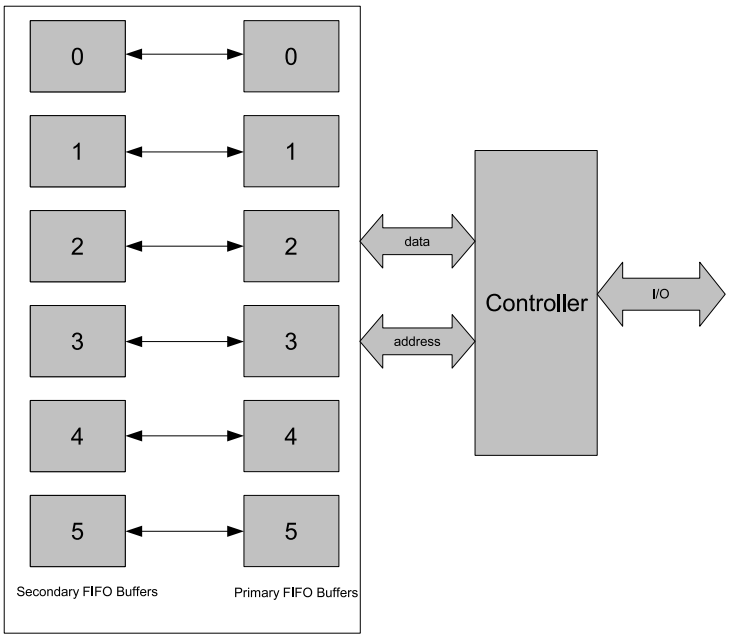

Fig. 3.3: Feeder FIFO data flow.

3.2.2 Distance Calculation

Distance calculation is responsible for determining the proximity of a pair of particles in three-dimensional space. The algorithm implements a Euclidean distance formula. A data flow graph representation of the algorithm is shown in fig. 3.4. In it, diamonds represent inputs and outputs to the distance calculator processing element. Circles represent mathematical computations and edges represent flow of data.

The algorithm takes as input the x, y, and z positions of two particles. It calculates the difference of each distance dimension, squares each result, and then sums the three products. The Euclidean distance algorithm actually calls for taking the square root of this final sum. However, due to the fact that the square of the distance is used in a later computation stage, and that square root is an expensive operation to implement, we utilize the square of the distance for our radius comparison. Finally, this squared radius is compared to the squared cutoff radius. Particle pairs within cutoff radius are passed on to the next computational stage and those that do not fall within cutoff radius are discarded.

Due to the fact that distance calculation is performed for every particle pair in a

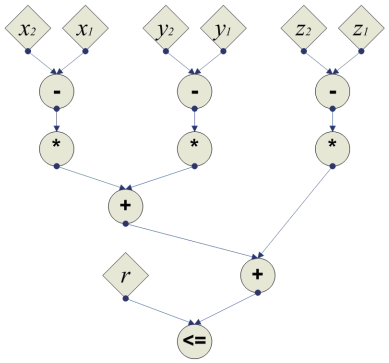

Fig. 3.4: Distance calculator data flow graph.

simulation, the hardware design of the DC unit is optimized for high throughput. Each mathematical operation shown in the data flow graph in fig. 3.4 corresponds to a mathematical unit in hardware. These mathematical units are instantiated components of Xilinx's floating-point IPCoregen utility, and each carries with it its own latency. This direct instantiation of the data flow graph makes use of fully pipelined functional units each with an initiation interval of one cycle. With such an approach, the DC architecture itself attains a net initiation interval of a single cycle.

The handling of data happens on the clock. The DC unit's two particles that it accepts as inputs consist of the candidate particle, to which all other particle locations are compared. These particles are designated by the triple $<x$, y, and $z>$ and a new particle for distance consideration is accepted every clock cycle. The latency of this processing element is 52 clock cycles and is derived from the critical path. The critical path commences with the x or y comparison. A multiplier takes 13 clock cycles, adders/subtracters 11, and the comparator 6. The discrepancy in the latencies for each mathematical unit is made up in delay units in hardware. For a substance with a low amount of particles within radius, the arechitecture throughput and efficiency will be largely determined by the throughput and efficiency of the DC unit. As will be reviewed in further detail later in the chapter, one of the goals of increasing throughput is instantiating several DC units to process particle pairs in parallel. A hardware level representation of the DC unit is shown in fig. 3.5.

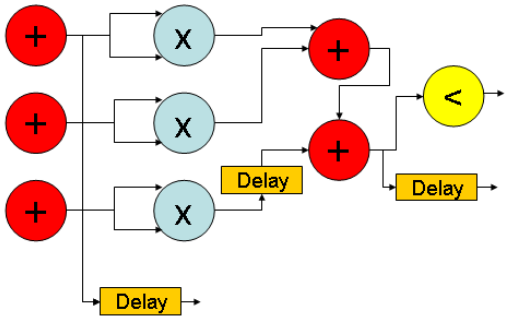

Fig. 3.5: Hardware-level view for the DC Unit.

3.2.3 Inverter

A hardware implementation of a floating-point inverter requires a significant number of chip resources, relative to other functional units. Consequently, as shown in fig. 3.2, it was developed separately from the distance and Lennard-Jones computational stages of the design. Coupled with the need to invert the calculated distances is the need to collect and queue computation results from multiple DC units operating in parallel, in preparation for the subsequent Lennard Jones Potential calculation phase. As such, we developed a set of FIFO buffers to hold the results of DC computation until they can be consumed by the LJPC units. This was done to provide a buffer for the data coming out of the distance calculation phase before it goes into the Lennard-Jones potential calculation stage. The Inverter unit is designed to receive the squared distance value resulting from distance calculation, invert it, and then queue the result in another set of output FIFO buffers for Lennard-Jones Potential calculation. The flow for this processing element is shown in fig. 3.6.

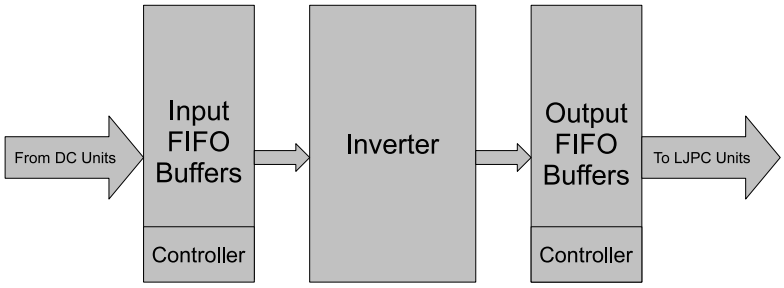

Fig. 3.6: Inverter unit with input and output FIFO buffers.

3.2.4 Lennard-Jones Potential

Lennard-Jones potential calculation, or LJPC as it is referred to in this paper, is a mathematical model and function that accurately describes the force between two atoms or molecules as a function of the distance between them. Because the distance is in all directions of three-dimensional space, it is also referred to as a radius. To illustrate and give some background to the forces being computed through Lennard-Jones, when two electron orbital clouds of two atoms come into contact, there is a strong repulsive force to drive the two atoms apart. This force is referred to as Pauli repulsion. At longer ranges, until the observed force is negligible at the cutoff radius, there is an attractive Van der Waals force. In 1924, John Lennard-Jones derived a mathematical expression that very closely describes these two behaviors for all radii. His equation, given in (3.1), can be used to derive the force between two particle pairs. It is this force equation, shown in (3.2), that is translated to hardware in order to compute the force between two particles. This force equation is derived by taking the negative of the gradient with respect to radius (r) of eq. (3.1).

In eq. (3.1), r is the radius between the particle pair. As such, Lennard-Jones potential is a function of radius. There are three radii of note which determine the magnitude and direction of the force between the particle pair. The first of which is the cutoff radius at and beyond which the force exerted is negligible. The attractive force between particle pairs increases as the distance shrinks from the cutoff threshold as it approaches a maximum attractive force represented by ϵ. From this radius and closer, the particle pair begins to experience a weakening attractive force due to the proximity of the electron orbital clouds. The Pauli forces cancel the Van der Waals forces at the radius σ and at that exact radius the net force is zero. All radii short of σ result in a repulsive force that increase exponentially in magnitude as the two particles' electron orbital clouds near one another. As the substance and chemical compositions change between MD simulations, these three parameters will change as well.

$$V(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}] \tag{3.1}$$

$$Force = 48\epsilon * \frac{\sigma}{r^2} * \frac{\sigma}{r^6} * (\frac{\sigma}{r^6} - 0.5) \tag{3.2}$$

Figure 3.7 illustrates the concept of the cutoff radius and the forces that result from the distances between particle pairs. Consider a candidate particle, particle x. Particle q lies within the radius designated by σ, and so the resulting force will be repulsive. Particle p lies beyond σ and within the cutoff radius r, so its force will be attractive and will attempt to draw the candidate particle x to it. This same operation will repeat for all particles in the system and the net force on particle x due to all neighbors will be stored for position and velocity update. Position and velocity data for particle x will not take place in the simulation until the net force between all particles is examined. To move particle x when the net force on it due to its neighbors is calculated would move it out of place and skew the results when it will be considered as a neighbor particle and particle p or q is considered as the candidate particle. Using Lennard-Jones calculation, the force between all particles as well as the total potential energy of the system can be derived. Of these two values, it is the total force on a particle that cause a change to its position and velocity.

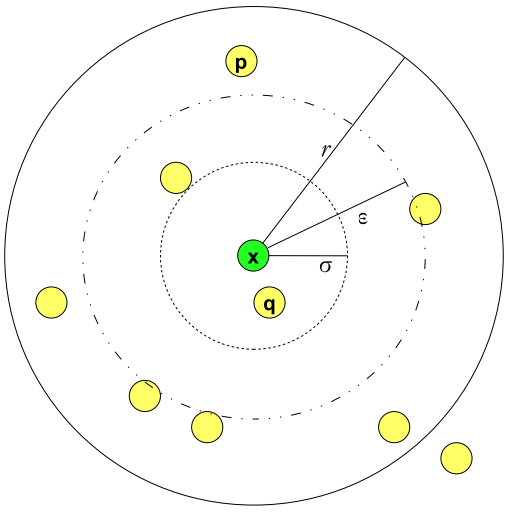

Fig. 3.7: Lennard-Jones cutoff radius.

The translation of eqs. (3.1) and (3.2) into a data flow graph is shown in fig. 3.8. The left hand branch of fig. 3.8 is the implementation of the force equation and the right hand branch is the implementation of the potential equation. Consider a contrast between the complexity of the Lennard-Jones data flow graph and the distance calculation data flow graph. It can be seen that the Lennard-Jones calculation is more complex and involves the use of more hardware resources than the DC unit does. Rather than making a direct instantiation of all mathematical operations as was the case for the DC unit, the Lennard-Lones Potential calculator (LJPC) unit mitigates the hardware footprint by dividing the algorithm into three sections and reusing the hardware between each section. Each stage has its own controller for hardware reuse. The hardware level including controllers is shown in fig. 3.9, where the muxed inputs are shown as part of each controller. Each controller has enough inputs and outputs to handle everything connected to it, but the VHDL code that drives the controller handles the muxing.

Section 1 begins with the inverted squared radius, $\frac{1}{r^2}$, which is retrieved from the output FIFO buffers in the Inverter computational stage. Section 1 has one multiplier and two subtractor units. The data is then passed to section 2 where three multipliers are used to accomplish all the hardware needs of that stage. Section 3 is the accumulator stage which contains four adders and holds the values for force in the x, y, and z directions as well as the total potential energy due to that force. Once the Lennard-Jones calculations have been completed, the data is held in temporary buffers for the next computational stage.

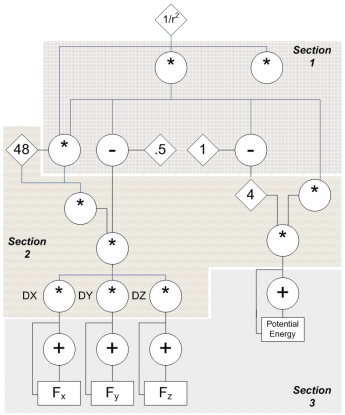

Fig. 3.8: Lennard-Jones potential calculator directed flow graph.

3.2.5 Verlet Integration

The force data derived in the Lennard-Jones calculation stage is used to update the position and velocity of each candidate particle. Relying on Newton's motion estimation equations, the Verlet method employs a method of integration to arrive at that goal. GROMACS [15] uses a derivative of the Verlet method known as the leap frog method in their MD simulations. It is this method that our project has implemented.

The leap frog method depends on sampling the velocity at staggered times to the position in order to compute next velocity and position. The staggered sampling rates mean that the velocity and position values leap over one another during the MD simulation. This is shown in fig. 3.10. The leap frog method is used in this MD project for two reasons. One, it is self starting. There is not a special case or extra method needed in order to derive initial positions and velocities. Secondly, it is time reversible. This is a property that minimizes the error drift that often arises in Euler integration methods. A more complete overview of the mathematics behind the leap frog method of Verlet integration is given by Berendsen and van Gunsteren [36].

The equations for the new velocity and position information for each particle are given in eqs. (3.3) and (3.4), respectively. For the time $\frac{-t}{2}$, we assume zero velocities. The *F(t)* term is what is derived from the Lennard-Jones computation stage.

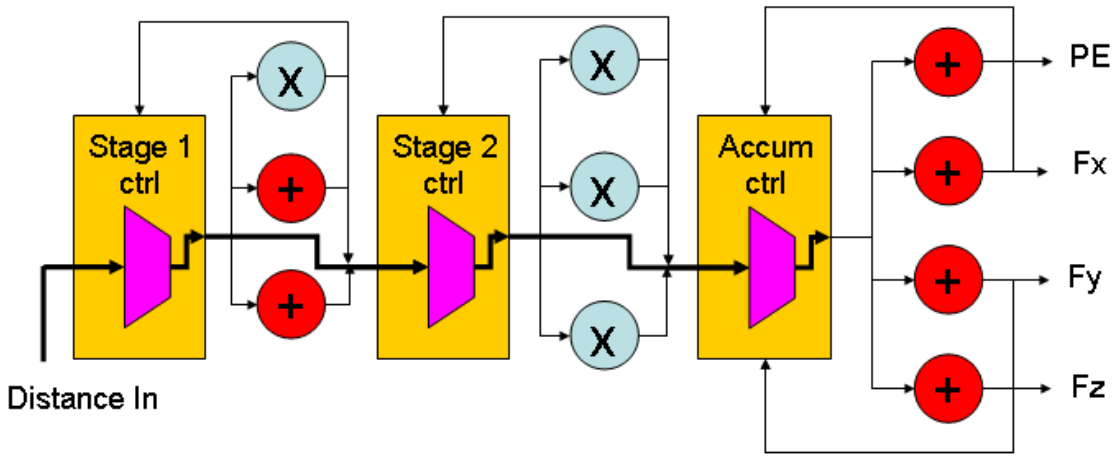

Fig. 3.9: Hardware-level view of LJPC unit.

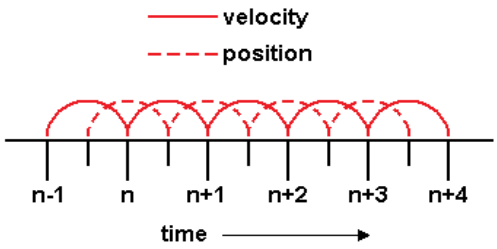

Fig. 3.10: Visual representation of the leap frog method.

$$v(t + \Delta t) = v(t - \frac{t}{2}) + \frac{F(t)}{m}\Delta t \tag{3.3}$$

$$pos(t + \Delta t) = pos(t) + v(t + \frac{\Delta t}{2}\Delta t) \tag{3.4}$$

3.2.6 Velocity and Position Update

Presented in fig. 3.11 is a simplified version of the directed flow graph for the Verlet unit. Because the force between two atoms is computed for the x, y, and z components, the architecture for the Verlet unit is done for each separate axis. Figure 3.12 is the hardware-level representation of the Verlet unit, where each resource depicted in the directed flow graph has a corresponding resource allocated to the processing element. The only difference between the directed flow graph and the allocation of processing elements is that the directed flow graph is replicated three times over so as to have each of the x, y and z components represented. This means that three times the amount of computation is required for Verlet update. Figure 3.12 shows this difference.

Also, it can be seen in fig. 3.11 that there are several inputs outside of the force. Because Verlet motion estimation integration equations are a function of time, the dt input represents the current time step. The mass, denoted by m, is a property of the particle. The *vel* and *pos* inputs are for the current velocity and position of the candidate particle. In an attempt to reduce the amount of storage mechanisms in the design, only the position is stored, while the initial velocities are assumed to be at rest. The next positions derived

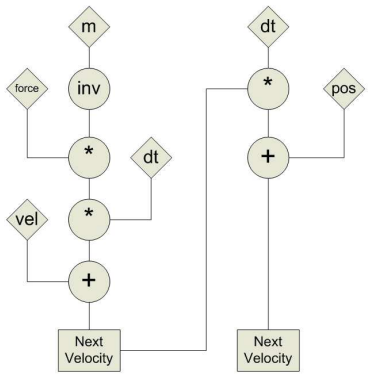

Fig. 3.11: Verlet unit directed flow graph.

are forwarded back to the Feeder unit's extra set of FIFOs, while the next velocities derived are latched and fed back into the Verlet unit, where they can be observed at any point in the simulation, but only for that time step.

Because the hardware called out by the directed flow graph is allocated to the processing element, there is no hardware reuse within the pipeline. In addition, all instantiations of Xilinx units are pipelined. The Verlet unit requires no formal controller for managing the functional units. However, some control logic was required to manage the latching and feedback of the computed velocity.

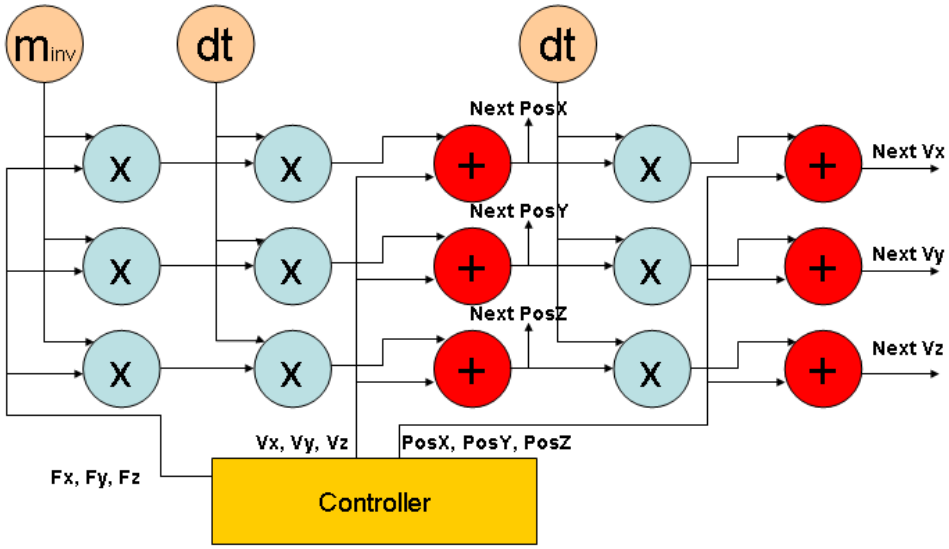

Fig. 3.12: Hardware-level view of Verlet unit.

3.2.7 FPGA Specific Architecture Design

To leverage the advantages of reconfigurable computing devices, we have selected the Xilinx Virtex4 family of FPGA devices as our target platform. The primary architecture features of particular interest to this project are spatial parallelism within the FPGA fabric, large aggregate on-chip memory bandwidth, and multiple configurable on-chip memories (BlockRAMs). We employ on-chip memories to implement inter-component buffering and FIFOs, which not only have the impact of decoupling the computational modules from each other for a more modular design, but also serve to balance dynamic changes in per-component production/consumption rates. Parallel designs that are modularized through rate matching and decoupling buffers are capable of high throughput, which we hope to prove in the following chapters. In the preceding subsections of this chapter, both the mathematical function as well as the hardware architecture realization of each processing element in the design were discussed. This section focuses on the integration of the various components into a system-level architecture for supporting dynamic load balancing, as well as the primary method of addressing the same. A conceptual view of system-level hardware architecture is presented in fig. 3.13.

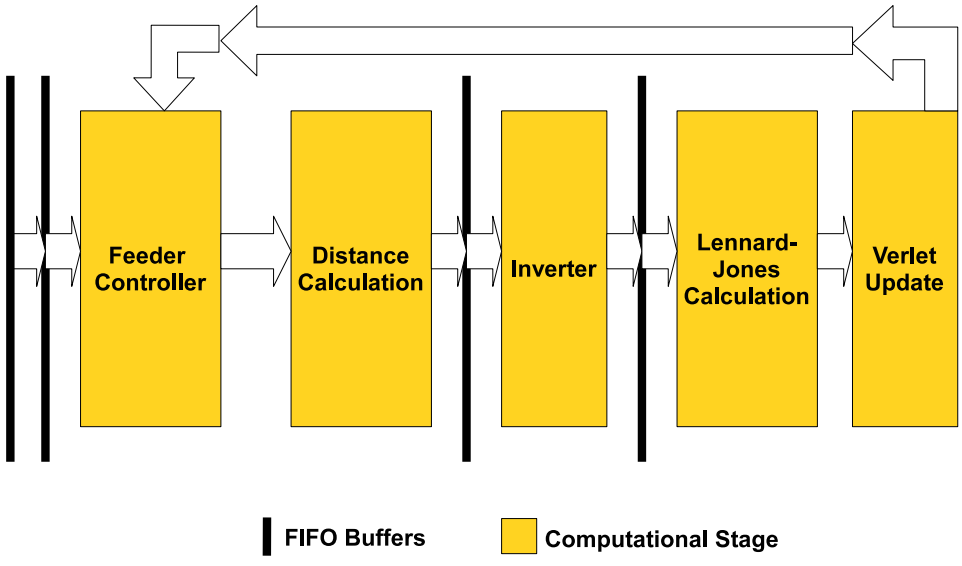

Fig. 3.13: System wide MD architecture data flow.

Figure 3.14 presents a general and basic hardware architecture which exploits some of the highlighted features of an FPGA such as spatial parallelism and on-chip buffering. Multiple instances of computational components are included in the design so as to increase overall computational throughput. Consumption/production rate mismatches are balanced through the inclusion of buffering between the DC components and the LJPC components. However, design challenges arise in deciding how many DC, LJPC, and Verlet processing units should be instantiated on the chip. While inter-stage buffering can resolve data rate mismatches to a certain degree, buffers have finite capacities, limiting the degree to which rate mismatches can be overcome. This imposes a strict limit to the level of load variance the architecture is capable of handling without significant decreases in resource utilization. Further, the ratio of DC to LJPC computations required for a particular simulation can vary dramatically from one substance to the next, and can even vary within a single simulation run. Drastic mismatches between the number of computations required and the computational resources dedicated to implementing those computations leads to inefficient use of the FPGA fabric, with under-utilization of some resources, over-utilization of others, and an overall impact of increased simulation time. However, what has remained unanswered is the question of how many units make a good design. The problem with Molecular Dynamics is that an engineer or scientist does not know how many particles will fall within the cutoff radius of a given candidate particle for every iteration and time step. This is the root source of the load imbalance in MD simulations. In order to alleviate the difficulty arising from load imbalance, we now introduce the Flex processer, which is a computational unit that can switch modes between distance and Lennard-Jones potential calculation. Such a unit provides for dynamic load balancing, increasing throughput and maximizing efficient use of area and resource utilization.

3.2.8 Flex Hardware Design

The Flex processor was first introduced in a previous work [37]. It can be configured to perform the functionality of either the DC or the LJPC module. Because of this, the architecture must offer sufficient resources to efficiently handle both data flow graphs. Using an

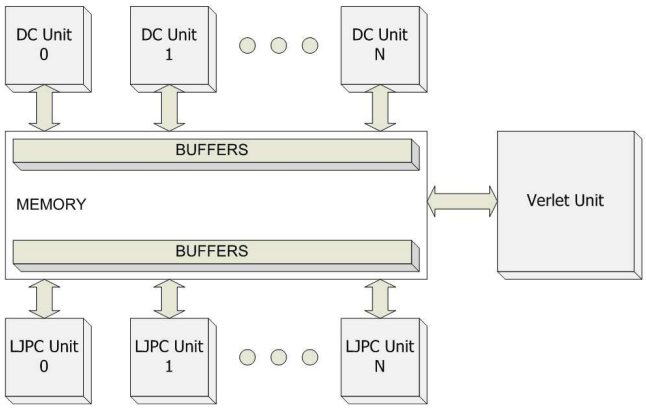

Fig. 3.14: Dynamic MD architecture.

iterative technique based upon force-directed scheduling, the architecture derived balances area consumption and throughput for both DC and LJPC computations. The final result of this investigation was a Flex unit that consists of three multipliers and five add/subtract units. The architecture is pure data flow, with delay registers inserted only at needed locations. This completely eliminates the need for register files and register file addressing. The DC operation is trivial, as there is a 1-to-1 mapping between nodes in the data flow graph and computational units in the processor. In LJPC mode, resources must be re-used as there are more nodes in the data flow graph than computational units. Extensive multiplexing is used to control data flow between the computational units. Arithmetic units can receive data from external buffers, stored constants, or other arithmetic units. The circuit requires multiplexors on each input port of each arithmetic unit to provide this functionality. This architecture is based upon the LJPC algorithm being unrolled seven times. Once the bulk of the calculations are performed, four different adder trees are employed to take care of the four accumulate operations which are the force in each direction (x, y, z) as well as the total potential energy of the candidate particle.

After analysis and debate, it was decided to use a 24-bit floating-point (8-bit exponent with 16-bit fraction) instead of a standard 32-bit floating-point size. This was done primarily to reduce the size of computational components on the FPGA to allow for everything to fit. The cost in loss of accuracy when compared to 32-bit floating-point is traded off with

improving gains in area consumption. Reducing the data size also changes the latency of some of the computational units.

The Flex processor is designed to execute one of two programs, computing either DC results or LJPC results. An instruction consists of a set of input selects for all multiplexors on a given clock cycle and for specifying addition or subtraction for each of the add/subtract units. A 16-bit instruction word is sufficient to handle these options. The processor must be able to switch between programs rapidly and without extensive code loading or storage. Because of this, the Flex program memory is constructed of individual ROMs, where each ROM is built of block RAM (BRAM) blocks. Each program is stored in a separate program memory. A multiplexer is used to determine which program is currently being executed. A hardware level view of the Flex unit including processing core, LJPC accumulator routing, multiplexer routing, and state-machine based controller is shown in fig. 3.15.

3.2.9 Dynamic Load Balancing, Decoupling, and Rate Matching

While the Flex unit is the primary method of addressing dynamic load balancing, the buffering capabilities of the FPGA and the advantages they offer facilitate the Flex unit's impact. This project refers to computational stages and the FIFO buffers that rest between them. It is these sets of FIFOs that allow each computational stage to operate with a lesser degree of dependence on one another. The stages are decoupled because when data is processed and sent from one computational stage to the next, it is stored in the FIFO buffers until such a time as it can be processed by the next stage. Because each stage can operate at different speeds, they are said to be rate matched. Rate matching and decoupling starts with the Feeder FIFO unit. As was mentioned previously, this unit contains two sets of buffers. Specifically, there are two sets of six buffers making 12 total FIFO units. One set, the primary set, is pre-loaded before runtime with the position data of all particles in the system. This set feeds the two DC units and four Flex units operating in DC mode. At the beginning of the simulation, there is no demand for Lennard-Jones potential calculation. The Feeder unit's controller sends data out to each DC unit. At an unknown time later in the simulation, one or all of the Flex units will have changed mode, leaving the original two

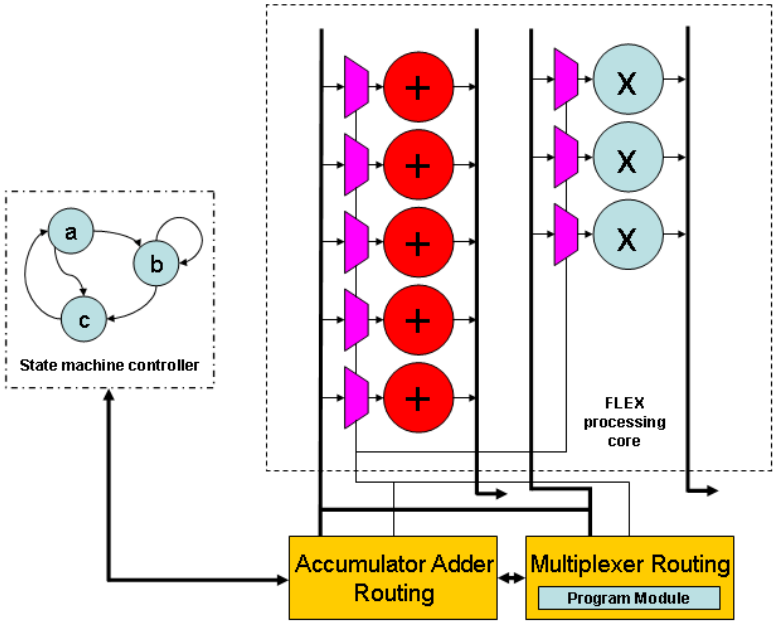

Fig. 3.15: Hardware-level view for Flex unit.

DC units operating on two of the FIFO buffers' data. When these two buffers are empty, the controller assigns the the DC units to process on the buffers with the most amount of data and the operation continues. If at any time a Flex unit switches back to DC mode, the controller assigns any of the unused buffers to it. This process is referred to as variable rate consumption. It addresses load balancing by always assuring that the buffer with the most demand for data processing is assigned a DC unit to it.

The design offers a second level of buffering, inserted between the DC computation stage and the LJPC stage. In sec. 3.2.3, we discussed the Inverter and briefly described the fact that it was coupled with buffering to support rate decoupling. The Inverter phase is composed of two sets of six FIFO buffers coupled to two inverter units.

Each set of FIFO buffers around the inverter unit receives data from the DC units (both DC proper and Flex) and send data to the LJPC units (both LJPC proper and Flex), respectively. This makes for three stages to the Inverter unit; the intermediary buffers between the DC units and Inverter units, the inverting operation itself, and the intermediary buffers between the Inverters and the LJPC units.

The buffers on the side of the Lennard-Jones potential calculation phase are linked in a cascading manner. When the first buffer reaches a high watermark level of 128 tokens of data, it begins to fill the next buffer, and so forth until the last buffer. When the final Output FIFO buffer is full, it sends a signal to the Inverter unit controller which proceeds to fill each Output FIFO to its capacity, beyond the high water mark. The Inverter unit's controller has a polling routine that checks to see if there are data in any of the Input FIFOs in order to ensure that no processing units idle needlessly. This is shown in fig. 3.16.

3.3 Control of Data Flow

Part of the architecture is the method by which the controllers operate. Data starts in the Feeder FIFO buffers, passes to the DC processing elements (both DC unit proper and Flex unit in DC mode), then to the Inverter and its buffers, next to the LJPC processing elements (both LJPC unit proper and Flex unit in LJPC mode), and finally to the Verlet update stage where it is processed and returned to the Feeder FIFOs to recommence at the next time step.

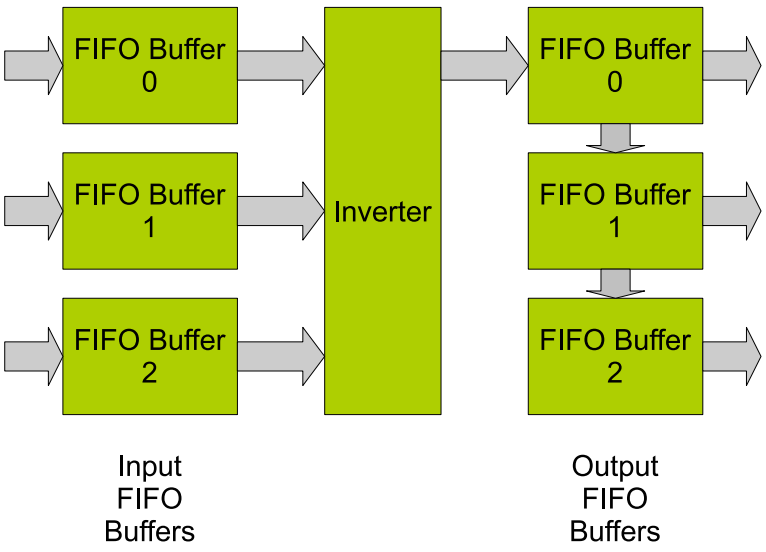

Fig. 3.16: Inverter input and output FIFO buffer configuration.

3.3.1 Feeder FIFO Data Flow

The controller in the Feeder FIFO unit determines which DC processing element processes data on which Feeder FIFO buffer. The six FIFO buffers are numbered 0 - 5, the DC units 0 and 1, and the Flex units 0 - 4. Feeder FIFO buffer 0, 1, and 2 are initially assigned to DC unit 0, Flex unit 0, and Flex unit 1, respectively. Feeder FIFO buffer 3, 4, and 5 are initially assigned to DC unit 1, Flex unit 2, and Flex unit 3. These assignments remain fixed until either the DC units finish processing data or until the Flex unit is commanded to switch modes. At both of these junctures, the controller determines the new assignments between processing elements and buffers. Feeder FIFO buffers with the most amounnt of data are assigned to the DC units whenever possible.

When Flex units switch away from DC to LJPC mode and then back, they will be assigned to any available Feeder FIFO buffer. While it is possible that a Flex unit will be assigned to a buffer with the most amount of data, this behavior was not observed in our simulation due to the fact that demand for LJPC processing causes the Flex units to switch modes and remain in LJPC mode. This is also due to the fact that Lennard-Jones potential computation takes more time and so the DC units fill the Inverter buffers faster than they can be emptied. Chapter 4 will discuss this in more detail.

3.3.2 Distance Computation Output Flow

The output of the DC processing elements is dictated by the main controller. The inverter stage was divided into two sets of an inverter unit, three input FIFO buffers, and three output FIFO buffers. This makes for a total of two inverter units, six inverter input FIFO buffers, and six inverter output FIFO buffers. It was decided to dedicate a DC unit and two Flex units to each portion of the inverter stage. DC unit 0 and Flex units 0 and 1 are directed to output to the first set of inverter input FIFO buffers, and DC unit 1 and Flex units 2 and 3 are directed to output to the second set of inverter input FIFO buffers.

The main controller monitors the levels of the Inverter output FIFO buffers and these buffers dictate the need for the Flex units to switch mode to process in LJPC mode. The Inverter output FIFO buffers fill in a cascading fashion where the last of the three in a

set directs its output to an LJPC unit. The data flow is done in the two sets of three Inverter output FIFO buffers. Each set of three has two Flex units and one LJPC unit allotted to process data on them, with the Flex unit processing on the two buffers that send their data one through the other into the third. The pre-determined threshold level which commands the Flex units to switch mode to LJPC mode is 128 data tokens, referred to as a high watermark. When the other two Inverter output FIFO buffers reach this level, the corresponding Flex unit switches and begins processing data. If it finishes processing data and there is still a need for distance calculation, it switches back.

This kind of decoupling between the inverting, distance calculation, and Lennard-Jones potential calculation phases of the design allow the all processing elements to continually work without bottle-necking around the input and output of the inverter units.

Chapter 4

Molecular Dynamics Results

The beginnings of this project were to provide an architecture that was capable of dynamic load balancing. The goal as it stood then was to use the Flex processor as a means to the end of accelerating an MD simulation. The goal of the project now is to prove the usefulness and impact of the Flex unit on the MD simulation. This includes observing the measures taken to ensure dynamic load balancing and quantifying them. Both hardware and software had been designed and generated to produce these results, where the hardware was the architecture and the software was an application specific architecture simulator. The focus of using the software is observe and monitor the results of the MD simulator, without having to resort to waveform-based analysis and long execution times encumbered with traditional HDL simulation tools. The following chapter will deal with how well those results compare with competing designs.

4.1 Hardware Results

The MD simulator in its final computational state consisted of a feeder controller, two DC units, two LJPC units, four Flex units, two Inverter units, and the Verlet unit. This design can be seen in fig. 4.1.

The Xilinx ISE place and route results are given in table 4.1 and represent the hardware footprint on the Xilinx VFX140 chip. It is relevant to note that although this is a dynamic architecture, the hardware footprint remains unchanged. The Flex processor through its program ROM and a series of MUXes is able to switch between processing element modes; both DC and LJPC mode. This means that we do not need configuration-plane-level reconfiguration in order to support the dynamism that the Flex architecture offers. Despite this advantage, there is a cost involved in the amount of hardware necessary to support the

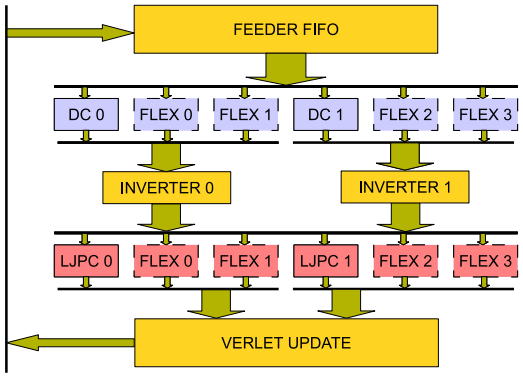

Fig. 4.1: Top-level MD system architecture.

Flex. Since all of the Flex architecture's resources are not in constant use, it is not perfect. However, the benefits it brings as far as dynamic load balancing is concerned are positive and will be reviewed more fully in Chapter 5.

The timing results from the synthesis showed that our design's maximum speed of 117MHz is attainable. This speed is not the final indication of proof of concept; Chapter 5 will tackle the subject of what proves the validity of using a Flex based design over a static one. As will be further touched on in the future work section, the advent of faster and more sophisticated FPGAs will only increase this ceiling, as well as offer a variety of other advantages for increasing the speed, throughput, and efficiency of this design. As reported in table 4.1, there is still room on the FPGA for other resource allocation if the need arises. More customized options for synthesis and place and route are also available that could free up more room on the chip and allow for more processing elements to be used.

Table 4.1: Hardware report for the Virtex4 VFX140 chip.

Number of Occupied Slices:	49654 out of 63168 (78%)
Number of Slice Flip Flops:	65067 out of 126336 (51%)
Number of 4 input LUTs:	69361 out of 126336 (54%)
Number used as logic:	53181
- Number used as Shift registers:	21966
- Number of FIFO16/RAMB16s:	232 out of 552 (42%)
- Number used as RAMB16s:	232
Number of bonded IOBs:	579 out of 768 (75%)

4.2 Hardware-Specific Software Simulator

In this section, data is provided to demonstrate the performance of the dynamic load balancing architecture under different circumstances. We first discuss the simulation-based approach to performance evaluation. Next, two groups of plots will be presented: plots representing resource utilization and plots representing FIFO levels at various stages in the design. Each plot will be discussed in detail highlighting the impact that particular element of the design has on throughput and load balancing.

The simulation that was run to produce the results in this chapter used a cutoff radius of 800 for Lennard-Jones computation. For our simulation, we do not use parameters that are specific to any particular substance. From the perspective of performance evaluation, the exact values of the simulation parameters do not have an impact on the effectiveness of load balancing. They would impact the degree of the load imbalance, but the controlling schemes that monitor data levels and signal the Flex processor to change mode are independent of these conditions. This means that the total time to simulate is not affected by these constants. In other words, for this architecture simulator, a simulation of a thousand salt water molecules would illustrate the same impact of the Flex processor and rate-matching techniques as would a simulation of 1000 noble gas atoms. The timing results would be drastically different, but the behavior between the static and dynamic architectures (as explained in the next chapter) would be demonstrable in both cases.

4.2.1 Hardware Simulator

A hardware simulator was employed to observe the behavior of the Flex based MD architecture. It contains modules that simulate each processing element and a controller that dictates data flow between them. Operations happen on a simulated clock, making it cycle accurate.

The simulator models the behavior of the architecture, and so begins and ends where the hardware begins and ends. A series of nested loops monitor the data as it goes from the feeder FIFO module to the LJPC module. When the LJPC modules have finished processing their data, the Verlet unit begins processing its data and when completed, over-

writes the position data and the time step is repeated with the new particle location. The computational states of each processing element are monitored throughout the simulation. It is these states that will be evaluated in the following sections.

Each hardware report given in this and subsequent chapters uses a composition of Slice Flip Flops, look up tables (LUTs) and block RAMs (BRAMs) to provide an idea of how much of the FPGA fabric was used.

4.2.2 FIFO Hardware Report

Each FIFO buffer is instantiated from the IPCoregen utility used within Xilinx ISE. The FIFO buffers are read and write enabled, making data available as soon as the consuming processing element makes the request. Table 4.2 shows the place and route report for the Inverter and Feeder units, which contain the Feeder FIFO, Inverter Input and Inverter Output FIFO buffers.

4.2.3 DC and LJPC Unit Utilization

The data shown in figs. 4.2 and 4.3 shows when the DC and LJPC units are active, and what computational state they are in while they are active. The plots only reflect the results through one iteration; that is, one candidate particle compared with the rest of the particles. The utilization of each of the Flex units will be shown by pie charts of the same nature later in this chapter. The pie slice labeled *busy* represents the time when the pipeline of the processing element does not allow it to accept new data to process. The pie slice labeled *working* refers to when new data is accepted and processed, and the idle time reflects when the processing element is not in use at all. Idle times occur when there is no new data to be processed and buffers are empty and waiting to be filled, or when the buffers are empty at the end of a simulation time step.

Table 4.2: Hardware report for the Feeder and Inverter units.

	Slice Flip Flops	LUTs	BRAMs	DSP48s
Feeder	165	693	144	7
Inverter	1158	1158	36	0

Table 4.3 shows a place and route report for the DC and LJPC units, as well as their latency in terms of computational cycles for the architecture. The mathematical units that are IPCoregen instantiations are optimized for look up table (LUT) usage rather than digital signal processor 48 (DSP48) usage. The IPCoregen utility is where each mathematical unit is configured to have the custom bit-width for floating point operations. The following sections discuss the utilization, or working versus idle time, of the DC and LJPC processing elements.

Figure 4.2 shows the resource utilization of the two DC units. The two DC units have identical architectures and delay times irrespective of the particle locations they are fed. Because the FPGA allows them to operate in parallel, the rate at which they process data will be identical. Each DC and Flex unit is initially assigned a FIFO in the Feeder FIFO unit. When a DC unit completes its data processing, the feeder controller assigns it a new buffer to process and allows the Flex unit that was previously processing data on that FIFO to work on a different FIFO if it is necessary to switch back to DC mode. There is no priority or fixed assignment. The Feeder FIFO controller assigns tasks to available processing elements as they become available.

The pie chart in fig. 4.2 does not show which Feeder FIFO buffer unit DC unit 0 and DC unit 1 are emptying. It reflects DC unit activity. The next group of plots that show FIFO levels will show that a Feeder FIFO buffer is emptied after the Flex unit attached to it switches modes. The DC units become idle when all Feeder FIFO buffers are empty. This means that the Inverter buffers, which receive DC unit output, were always able to accept data and that the DC units were able to function as long as there was data to be processed. The idle time comes near the end of the simulation, where the DC stage is complete and the more time-intensive LJPC and Verlet stage is still continuing their processing of data.

Table 4.3: Hardware report for the DC and LJPC units.

	Slice Flip Flops	LUTs	BRAMs	DSP48s	Latency
DC	2905	3039	0	1	52
LJPC	5180	5354	0	2	63

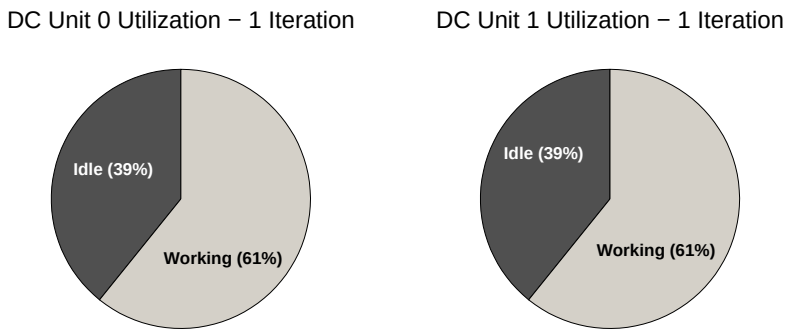

Fig. 4.2: DC units 0 and 1 utilization.

Figure 4.3 shows the utilization of states of the LJPC units. As their pipeline is the same length with the same latency, their behavior is identical. This is due to how the DC units send their data to the Inverter FIFOs. There are two Inverter units each with three input FIFOs. One of these FIFOs has its input from one of the DC units, and the other two from two of the four Flex units. This flow of data does not change, but the Feeder FIFO that the DC unit is assigned to does change. For distributions of particles that are uneven, there will be a tendency to even out as the DC units switch from one Feeder FIFO buffer to the next. The output from each DC unit will go to its respective Inverter FIFO and help distribute the load. As soon as the Inverter Output FIFO buffers begin to fill, the LJPC units begin processing that data. The same assignment of data paths exists between the Output FIFOs in the Inverter and the LJPC units. The idle time is so short in comparison to the DC units because as soon as data is available from the Inverter units the LJPC unit can begin its work. Because it is a more time consuming process, it takes longer to empty its Output FIFO buffers and so the LJPC units are often in use until the very end of the simulation.

The busy state of the LJPC unit is due to the partitioning of the resources into the three sections outlined in fig. 3.8. The LJPC unit can accept new data on each cycle for six clock cycles, after which it must process that data. No new data can be accepted for another 18 clock cycles, due to the nature of the hardware re-use within pipeline stages of

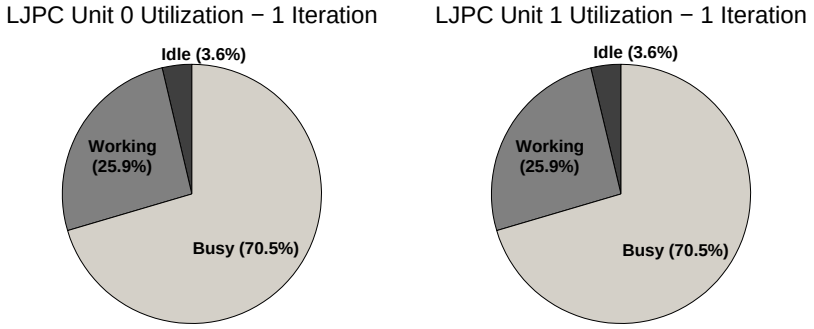

Fig. 4.3: LJPC units 0 and 1 utilization.

the LJPC.

4.2.4 Flex Unit Utilization

The next pair of pie charts shown in fig. 4.4 shows the behavior of two of the Flex units. All Flex units start the simulation time step in DC mode and are commanded to switch modes when the requirements of the system change. Each Flex unit is assigned to a different Feeder FIFO buffer by the controller unit. These assignments direct the Flex unit to retrieve data from the specific Feeder FIFO unit until a Feeder FIFO unit is empty. While the Feeder FIFO buffers have data, Flex Unit 0 is assigned to Feeder FIFO 2 and Flex Unit 2 is assigned to Feeder FIFO 3. The main controller selects Flex units 0 and 2 to switch to LJPC first, leaving Flex units 1 and 3 to stay in DC mode a bit longer. This allocation of Flex units 0 and 2 is reflected in the behavior observed in figs. 4.4 and 4.5. The idle time is negligible because each of these Flex units begins the simulation in DC mode and ends in LJPC mode. Barring a simulation where all the data happens to be handled by two of the four Flex units only, this means that at any point in the simulation, a Flex unit will be active. The same busy state associated with the LJPC units applies to the Flex units in LJPC mode. The purpose of tracking the computational states of the Flex units was to ensure our architectural features directly impacting load balancing are efficient. Table 4.4 shows the hardware report for the Flex and Verlet units.

Table 4.4: Hardware report for the Flex and Verlet units.

	Slice Flip Flops	LUTs	BRAMs	DSP48s	Latency
Flex	3261	3828	4	0	52 (DC) 83 (LJPC)
Verlet	5451	3454	0	0	53

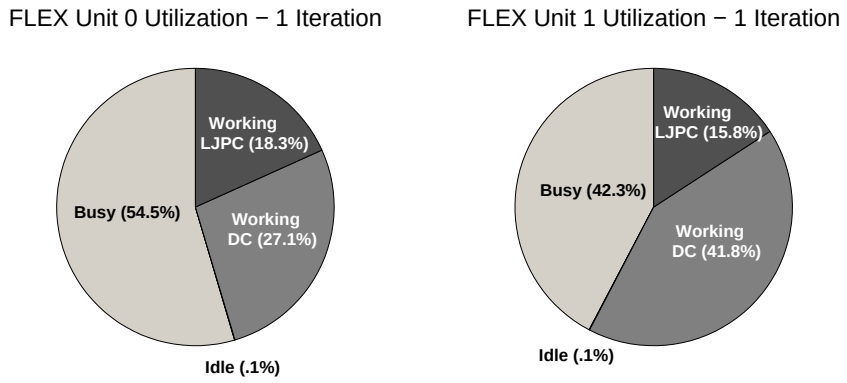

Fig. 4.4: Flex units 0 and 1 utilization.

The second pair of Flex units shown in fig. 4.5 have a similar behavior pattern to fig. 4.4. The reason for this is because the two DC units which are perfectly pipelined process data at identical rates, causing two of the Flex units to change in to LJPC mode at the time in the simulation when the DC units have emptied their assigned Feeder FIFO units.

4.2.5 Feeder FIFO Buffer Levels

Figure 4.6 shows Feeder FIFO levels from units 0, 1, and 2 as emptied by the DC 0, Flex 0 and Flex 1 units, respectively. The solid line shows a constant decrease in Feeder FIFO level, until the Feeder FIFO is empty. This is due to the fact that it corresponds to Feeder FIFO 0, which is associated with DC unit 0. As described previously, the DC units accept new data from their respective Feeder FIFO buffer on each clock cycle. This leads to a constant reduction in those FIFO levels.

The broken lines which represent the Feeder FIFO buffers emptied by the two Flex units share the same downward trend until just before clock cycle 500, when the level of

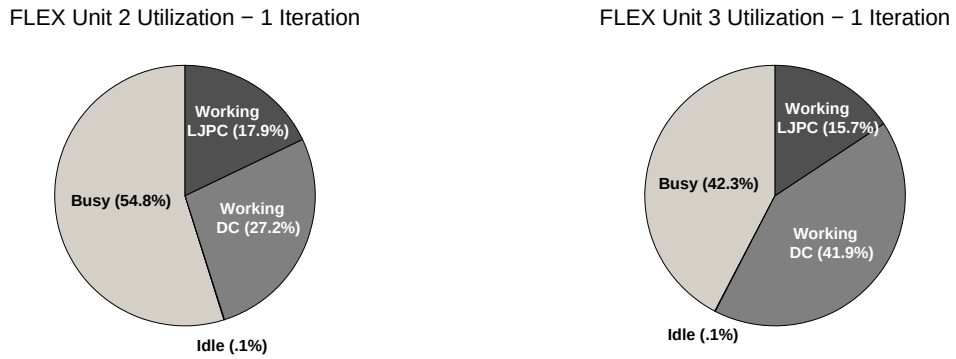

Fig. 4.5: Flex units 2 and 3 utilization.

Feeder FIFO buffer 1 holds constant for around 250 cycles. Before this transition point , the Inverter FIFO levels have not crossed the threshold that trigger a switch in Flex unit mode. When that threshold has been reached, the controller commands Flex 0 which is working on Feeder FIFO unit 1 to switch to LJPC mode. The controller commands Flex units 0 and 2 to switch to LJPC mode before it commands Flex units 1 and 3 to switch, which is why in fig. 4.6 Feeder FIFO buffer unit 1 is flatter for longer.

Figure 4.7 shows the levels in the Inverter input FIFO levels. The data in the input buffers are processed by the Inverter unit according to priority. When there is data in Inverter buffer 0, it is processed until empty. When empty, the Inverter unit checks for data in buffer 1 and begins processing data until it is empty or until there is data again in buffer 0. Buffer 2 is only processed when buffers 0 and 1 are empty.

Around clock cycle 500, Inverter input buffer 0 stops filling on every clock cycle. This is illustrated by the downward slope of the levels in Inverter input buffer 1. The data in buffer 0 is emptied, and then buffer 1 is processed, and then back to buffer 0. This is why buffer 0's level do not show any data in them. Inverter input buffers 1 and 2, receiving input from Flex units 1 and 2, respectively, cease filling when the Flex units switch back to LJPC. The data levels in buffer 2 remain static around clock cycle 750 until around 1350 for this reason. Once buffer 0 is empty and receives no more input at all from the DC unit affixed to it, the Inverter units focuses on buffer 1 until it is empty and then on buffer 2 until it is empty.

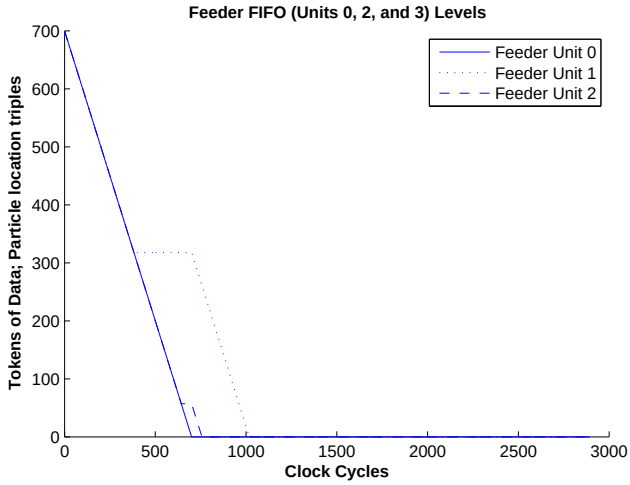

Fig. 4.6: Rates of data consumption for Feeder FIFO units 0, 2, and 3.

Figure 4.8 shows the Inverter Output FIFO buffer levels. These buffers fill in a cascading manner, from Inverter buffer 0, to Inverter buffer 1, and then to Inverter bufer 2. Buffer 0 is filled to a high watermark threshold of 128 data tokens and then the output of the Inverter begins to fill buffer 1 until the same threshold of 128 data tokens. At this point buffer 2 is filled until the levels in buffer 0 or 1 change. The main controller monitors these levels and communicates with the Inverter unit's controller to direct this flow of data. This was setup due to the fact that the design cannot rely on the Flex units to remain in LJPC mode. The dedicated LPJC processing element, the LJPC unit, will always have the most full buffer without the cost in controller overhead.

This behavior is reflected in fig. 4.8. The broken line which represents the levels of buffer unit 2 are attached to the LJPC unit. The other two lines, the solid and dotted, represent buffers 0 and 1 which are connected to Flex units 0 and 1, respectively. Buffer 0 fills to the threshold level of 128 and then holds steady. Buffer 1, as soon as buffer 0 is at the threshold level of 128, begins to fill to the same level. When buffer 1 reaches its threshold level, buffer 2 begins to fill with the output of the Inverter unit. Buffer 2 fills faster than the LJPC unit attached to it can process data and so it continues to fill until the Inverter unit completes processing the data passed to it by the distance calculation stage. This happens around clock cycle 2000. In most cases, the curve is jagged. This is occurs when the Flex

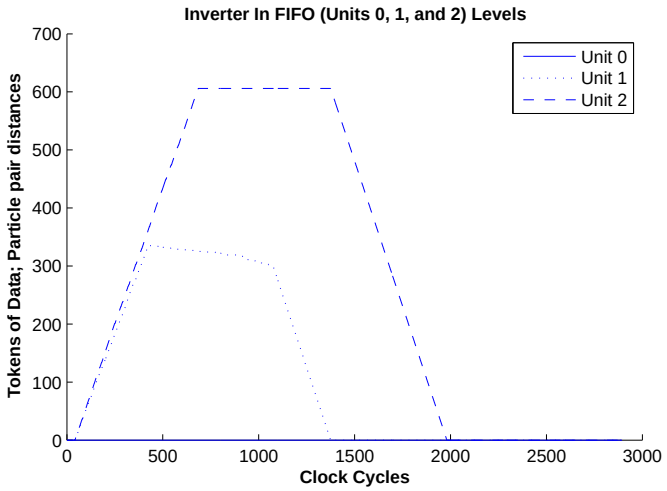

Fig. 4.7: Rates of data consumption for Inverter input FIFO units 0, 1, and 2.

units are in LJPC mode and begin to process data. The threshold level is no longer met, and the controller redirects traffic. The threshold level is reached again, and data begins to go back into buffer 2. By that time, a few clock cycles have passed and the LJPC unit is able to decrease the level in buffer 2 before it refills. This continues until around clock cycle 2000, when the buffers cease filling and the data has passed all the way through the distance calculation and inverter calculation phases.

The next group of plots discussed show how examining two different candidate particles can result in a different amount of total atoms within cutoff radius. This difference in particle pairs within radius per iteration illustrates the necessity for dynamic load balancing and highlights the unpredictability of an MD simulation.

Figure 4.9 shows the level in Feeder FIFO buffer 2 between two different iterations or when two different particles are considered as candidate particles. This plot reflects the difference in how long Flex unit 1 remains in DC mode between two different iterations. To recall, the buffer level empties when a distance calculating processing element is attached to it and this is reflected by the downward slope. The flat slope represents a time when Flex unit 1 is no longer operating in DC mode and the DC units are still busy with their own Feeder FIFO buffer unit. In the case of iteration 980, there are more particles within radius than on iteration 0. Consequently, the Inverter Output FIFO buffer fills faster in

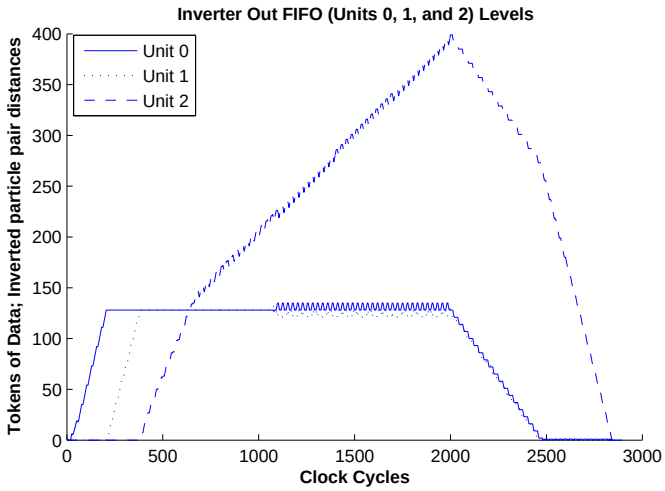

Fig. 4.8: Rates of data consumption for Inverter output FIFO units 0, 1, and 2.

iteration 980 than in iteration 0, causing the Flex unit attached to Feeder FIFO Unit 2 to switch from DC to LJPC mode earlier in the simulation than at itertion 0. This causes the flat plateau to occur earlier in the plot, which means in iteration 980 the amount of data left in Feeder FIFO buffer 2 is more than in iteration 0. It also means that the flat part of the curve for iteration 980 lasts for a longer time than iteration 0.

The flat part of the curve remains until either the Flex unit can switch back, or until the DC unit can take over processing data. In both iterations, it is the DC units that take over before the Flex unit switches back. Because the DC units have no stalls in the pipeline, the amount of time it takes for them to process the data in their corresponding Feeder FIFO buffers remains static from iteration to iteration. This means that the DC units that take over processing data on Feeder FIFO buffer 2 begin processing data on buffer 2 at the same time between these two iterations. Thus, the processing restarts and the flat curve trends downward again at the same time. The difference in time to empty corresponds to the different amounts of data left in buffer 2 by the changing Flex unit.

These small differences in particle pairs can have a large impact in time needed to compute force through Lennard-Jones potential operations. Figure 4.10 shows the difference in Inverter Output FIFO buffer levels on iteration 980 and iteration 0.

The time difference in the Flex units switching modes also has an impact on buffer

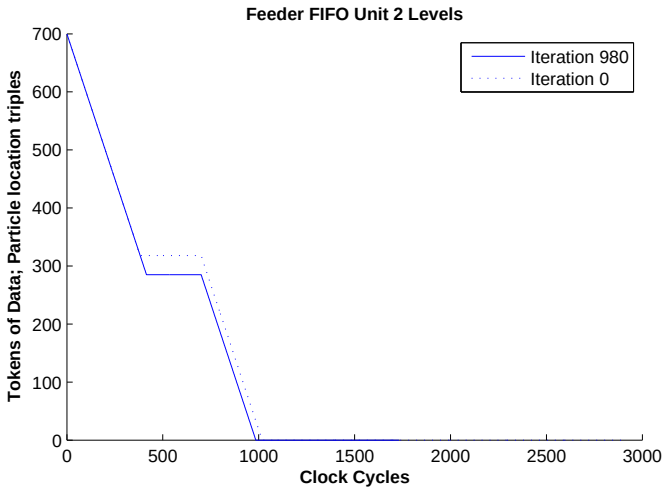

Fig. 4.9: Rates of data consumption between two iterations for Feeder FIFO unit 2.

levels for the Inverter output FIFOs. In fig. 4.10, the difference in Inverter Output FIFO buffer levels on iteration 980 and iteration 0 is illustrated. This buffer, to recall, is only emptied when LJPC unit 0 processes data. It fills when buffer 0 followed by buffer 1 reach the threshold level of 128 data tokens. It continues to fill while that threshold remains. If Flex units 0 and 1 switch modes to LJPC mode, then they will begin to process data on Inverter output buffers 0 and 1, drawing the data level below the threshold of 128 data tokens.

If they stay in LJPC mode, they will keep the levels down in their corresponding buffers, slowing the waterfall fill method and keeping the levels in buffer 2 from filling so high. In iteration 0, the Flex units switched much earlier in the simulation and begin to processon buffers 0 and 1, which stop the cascading fill action. This is another result of the Flex unit switching earlier. Figure 4.9 shows its line representing iteration 0 stop emptying the Feeder FIFO buffer 2 earlier than iteration 980. In fig. 4.10, the Flex units that switched stopped the cascading fill action which meant that LJPC Unit 0 processed the data on a buffer that continue to fill faster than it could be emptied.

All these figures illustrate the impredictability of a given set of particles for a given candidate particle. It should be noted that the plots presented in this chapter do not show how addressing dynamic load balancing increases throughput. Instead, the plots presented

in this chapter reflect the need to address load balancing. Were it not for the Flex unit, the FIFO buffer levels would show a lengthened amount of time where each FIFO or buffer is full.

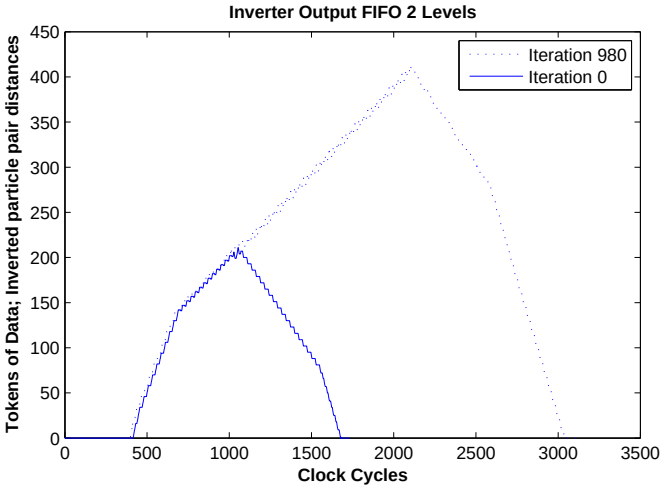

Fig. 4.10: Rates of data consumption between two iterations for Inverter output FIFO unit 2.

Chapter 5

Architecture Simulator Development and Testing

This chapter addresses the development and testing of competing static architectures to isolate the impact of the Flex processor on the performance of the MD architecture simulator. Specifically, the performance is rated by total processing time, throughput as defined by particles processed per second, and efficiency as defined by throughput per area. Testing is accomplished by the creation and benchmarking of three static architectures whose configurations differ in that there is no Flex processor, and the layout of the DC, LJPC, and Verlet units differ. The testing parameters and results are discussed and compared with the Flex-based dynamic architecture.

5.1 Static Architecture Simulators

Each static architecture simulator was created by first assessing three typical load distributions of particles within radius. These loads can be categorized as heavy, medium, and light. A heavy load is described as having a high number of neighbors within cutoff radius, where the LJPC units and FIFO buffers that use their data are consistently in use and consistently full. Light loads can be described as having low percentages of neighbors within cutoff radius, where the LJPC contribution is light because there is no demand for Lennard-Jones potential calculation. In heavy loads, the time to simulate is driven by how long the LJPC and Verlet units complete their tasks. In light loads, the DC units account for the majority of clock time. Medium loads are a split between heavy and light, and for the purposes of these tests, constitute a range of particles within radius at close to 35% - 65% of neighbors within cutoff radius. Each static architecture simulator describes a behavior pattern that can be mimicked by the Flex-based architecture. All four architecture simulators are benchmarked at a range of particles within radius from 7% to

94%. Dividing this range up into equal thirds will show in later sections of this chapter that the light load architecture simulator will perform comparably to the Flex-based architecture simulator because for those loads they will have a similar, if not identical, configuration of architecture. The same follows for the medium and heavy loads.

The makeup of the four architectures is given in table 5.1 and their configurations are shown in figs. 5.1, 5.2, and 5.3. The configurations are important to the testing and results. As will be explained in further detail in the following sections to the chapter, each architecture attempts to assume a predicted load balance based on the numbers of particles within radius for the life of the simulation.

5.1.1 Architectural Details

Each architecture simulator is a discrete event simulator with a single global clock where the events are the computational stages of MD. The conditions that dictate which events occur are the inputs and outputs of particle pairs. Incrementing the clock may trigger reactions from the simulated hardware modules. Each are checked in turn to allow those with pending reactions to proceed. Once all modules and storage elements have been given the opportunity of proceeding, the clock is incremented and the process repeats for a new particle pair. While the architecture proceeds by stepping through the clock cycle by cycle, much information is logged to output files, such as current FIFO levels, indicating the performance of the simulated circuit and behavior. Final outputs of the architecture include calculated resource allocation, total time required to complete the MD simulation run by the simulated architecture, and output FIFO levels. These data values were used to plot the results seen later in the chapter.

Table 5.1: Architecture configurations.

1. Architecture A - 6 DC units, 2 LJPC units (Static, light loads)
2. Architecture B - 2 DC units, 6 LJPC units (Static, heavy loads)
3. Architecture C - 4 DC units, 4 LJPC units (Static, medium loads)
4. Architecture D - 2 DC units, 2 LJPC units, and 4 Flex units (Dynamic, all loads)

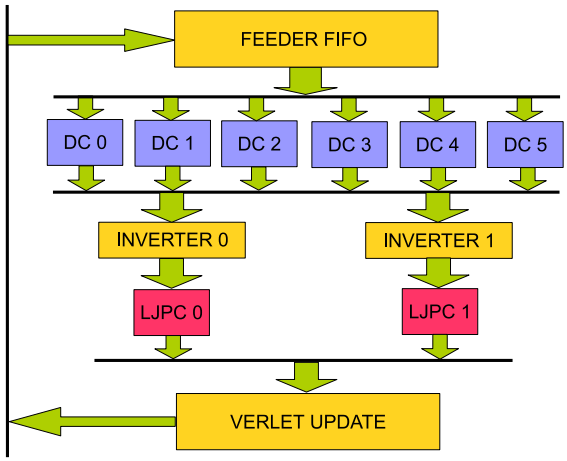

Fig. 5.1: Static architecture simulator A.

Each architecture simulator is structured to perform each fundamental operation (floating-point addition, subtraction, etc.) using native functionality, but to simulate the latency of hardware functional units through the insertion of clock-based buffering. The buffering models hardware shift registers, and is implemented as a storage element with an associated software based counter, which decrements based on the simulated clock counter. Once the simulated shift register's clock reaches 0, the value contained is passed to another element in the processing chain. FIFOs connecting the various modules are simulated as well, using storage elements and counter variables. The simulation of the controller monitors each of the FIFO levels through a polling routine and in the case of Architecture D commands the Flex processors to change modes when appropriate. These events repeat at varying frequencies until the simulation terminates.

5.1.2 Automated Architecture Simulator Generation

During the course of the development of the Flex-based architecture, a fellow student at Utah State University had developed a program that would derive an architecture and schedule based on a text-based directed flow graph and FDS scheduling routine. Named CHARGER, it provided a quick way to develop architectures for the processing elements such as the DC, LJPC, and Verlet units. The appeal of automatically generating the pieces

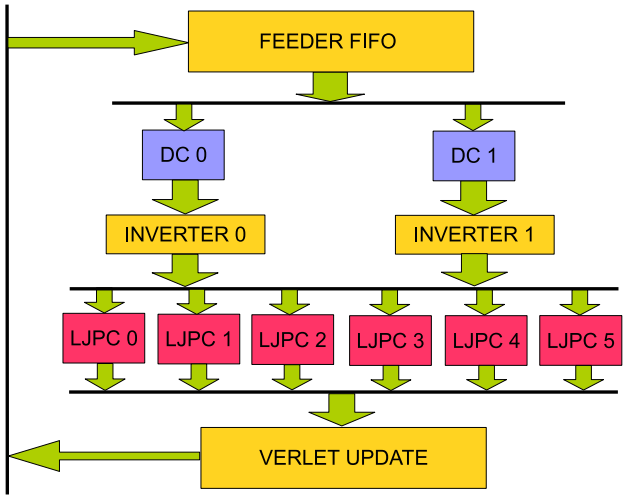

Fig. 5.2: Static architecture simulator B.

of an architecture and then putting them together manually led the project down that path.

CHARGER's impact on the project led to significant amounts of work done in two areas, the first being the software toolkit of CHARGER itself. CHARGER had limitations and bugs, most of which surrounded the method of feeding of input to each CHARGER based module. In addition, there was little success in producing a processing element that was architecturally similar to what had already been designed. CHARGER's strength of re-using hardware made comparing a CHARGER-based architecture to the Flex-based architecture difficult. The results had to be interpreted through the lens of what each architecture was composed of, how the hardware was being used, and whether or not that made an impact on the results.

The results of the automatically generated static architectures could not be explained without assumptions that compromised their integrity and in the end the use of CHARGER was abandoned.

5.2 Comparative Testing

The cost of adaptability offered by the Flex processor is in controller overhead and mode tracking. The mix of processing elements in each static architecture should reflect the predominant computational load. If nothing else, the dynamic load balancing architecture

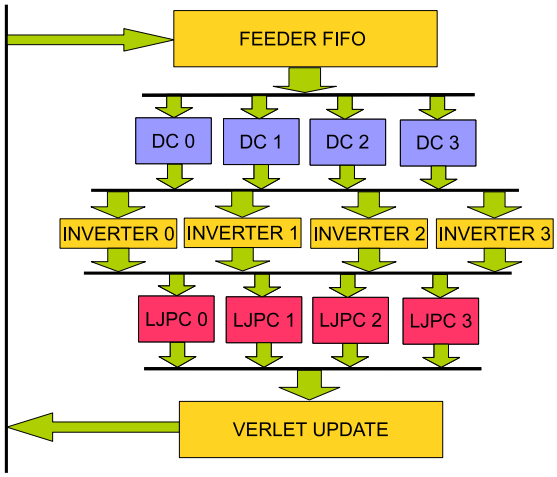

Fig. 5.3: Static architecture simulator C.

alleviates the minutiae involved in enumerating different architectures for varying loads. Using static architectures and evaluating their results against the results already gleaned from the dynamic architecture, it is possible to evaluate the Flex processor's contribution to MD simulation performance, and to examine the impact of the hardware overhead penalty incurred by its integration. Only by jointly evaluating simulation performance and area required to achieve that performance can we determine the real effectiveness of the Flex processor.

Each test was run on the same set of random location data for a set of 1200 and 3600 particles, where the cutoff radius was increased over the previously mentioned 7% to 94% of particles within radius. This range of percentage of particles in radius does not reflect an expected or predicted set of MD results. Rather, this range was selected to attempt to fully explore the behavior of the architecture simulators.

5.2.1 Testing Parameters

The most important testing parameter is efficiency which can be measured by throughput as a function of area used. Area, for the purposes of efficiency calculation, is derived by taking the total numbers of Slice Flip Flops, LUTs, DSP48s, and blocks of random access memory (BRAM) from the place and route results, and then adding up the decimal form

of the percentages used. This total percentage, which will range from 0 (empty - no hardware) to 4 (completely full, all Slice Flip Flops, LUTs, DSP48s, and BRAMs used) is taken together with the original total area and multiplied. The new derived area gives a sense for how much of the hardware is used and how costly the architecture is. The hardware reports given in tables 4.2, 4.3, and 4.4 in Chapter 4 are used to gather the data necessary to derive the new area. The thrust of exploring the area and throughput together as an efficiency matrix is this: if one MD architecture is capable of processing more particles in a lesser amount of time than another, then that design could be deemed more efficient. This information is shown in table 5.2.

However, raw processing throughput is not the only factor that must be considered, especially when targeting flexible platforms such as FPGAs. Throughput is defined for this project in terms of particles processed per second. To recall, each simulation time step for a MD simulator will contain the same number of particles, which means that the required number of particles that must be considered for computation does not change. Two factors may influence the rate at which these input particles can be processed: the first of which is the capability of the distance calculating stage. The second of which is the percentage of total particles that fall within cutoff radius. Thus, input throughput must be increased by adjusting the capabilities of distance computation stage. Because of the limited resources of the system, this adjustment will have an impact on the output throughput, which is the rate at which particles that fall within radius can be processed by the Lennard-Jones and Verlet computational stages. These particles are a subset of the input particles, which defines the relationship between input and output throughput. A good dynamic load balancing scheme will increase throughput on both fronts.

Table 5.2: Effective areas for each architecture.

Architecture	Area from Place and Route	Sum of Resource Ratios	Adjusted Area
A	80337	1.488	119568
B	142914	1.815	259489
C	111769	2.154	240845
D	120459	1.411	169929

Achieving high throughput computation typically comes at the cost of resource consumption, loosely referred to in terms of chip area. Exploiting the temporal and spatial advantages the FPGA offers can increase the overall throughput of the MD design. Throughput is calculated by taking the total number of clock cycles a given architecture takes to simulate a set number of particles, where a known number of those particles fall within radius.

5.2.2 Testing Results

The results between architectures A, B, and C illustrate the tradeoffs and impact resulting from changes to the configuration of processing elements. Comparing Architectures A, B, and C to Architecture D will highlight the importance of the Flex unit and its impact on throughput and efficiency. For testing purposes, one set of randomly generated particle locations was created, where each x, y, and z component falls between 0 and 500. All four architectures used the same set of data so that the results could be compared. The first step in testing was to measure the total number of clock cycles taken by each architecture to process all its data. These results are shown in figs. 5.4 and 5.5.

The results in figs. 5.4 and 5.5 show the total number of clock cycles necessary to complete one simulation time step for all four architectures, run with both 1200 and 3600 total particles in the system. This was done to further solidify the need for a dynamic design and to show that the behavior patterns of each simulated architecture are scalable. The x-axis of the plot represents an increasing number of particles within radius, and so it is expected that for low particles within radius, all four architectures have similar numbers for total clock cycles. For small cutoff radii, and hence low percentages within radius, each architecture has close to 250 total clock cycles. This is due to the fact that the maximum number of DC units for any architecture is 6, and the total number of particles in the system 1200, yielding 200 clock cycles to go through all 1200 particles. The remaining clock cycles are attributed to the latency of the DC unit (52 clock cycles) and the impact of the controller overhead. Similar effects are seen in fig. 5.5, though scaled to match the 3600 particles processed.

As the plot transitions into the middle range of cutoff radii, the differences in total clock cycles becomes more pronounced. Architecture A's lack of DC unit processing power begins to lag the simulation down and its total clock cycle number goes up considerably. Architecture C, which has two fewer DC units than Architecture B and two more than Architecture A also sees an increase in total simulation time. These trends continue to the right-most portion of both figs. 5.4 and 5.5, where there is a high number of particles within radius. At this juncture, it is Architecture B that performs best among static architectures, as it has a high number of LJPC units to handle the high demand of particles within radius.

It is worth noting that Architecture D, representing the Flex-based architecture, follows the behavior of Architecture A for small cutoff radii, Architecture B for large cutoff radii, and Architecture C for moderately sized cutoff radii. This is due to the fact that the Flex units meet the demands of the system, and change their modes such that at each juncture, Architecture D is functionally identical to the other three Architectures. This shows that the dynamic design is best with respect to execution time for any load; that is, any percentage of particles in radius.

Figures 5.6 and 5.7 present the simulation results from the perspective of throughput. Throughput was calculated as the number of particles processed per clock cycle, based on the 117 MHz clock frequency that Xilinx ISE PAR results reported.

The throughput values naturally decrease as a function of total percentage in radius. The more particles present in radius, the more time it takes for the architecture to complete

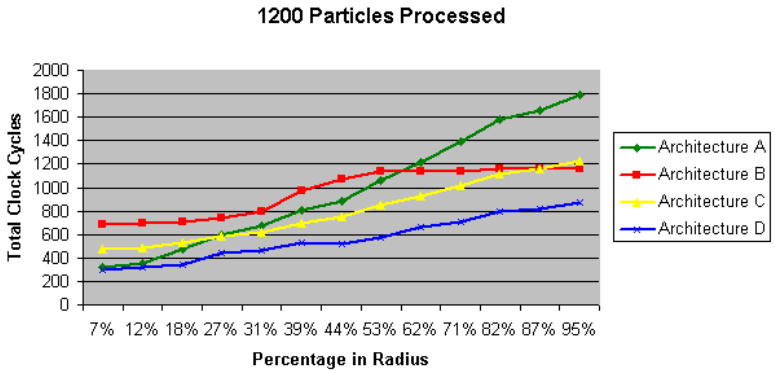

Fig. 5.4: 1200 particles processed.

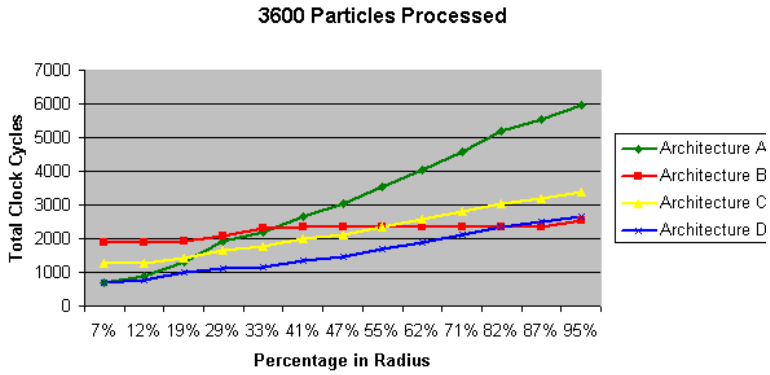

Fig. 5.5: 3600 particles processed.

the simulation, reducing throughput. It is interesting to note that Architecture B has a very flat throughput curve. This is due to the fact that for lower to middle range of cutoff radii, the total throughput is limited by DC computation. Lennard-Jones potential computation dominates for larger radii. Since Architecture B has two DC units and six LJPC units, the LJPC units are capable of keeping up with load increases until larger radii are reached on the right part of the graph. Where this implementation suffers is in resource utilization: for small to medium radii, the DC units are fully utilized, whereas the LJPC units suffer low utilization.

Efficiency, measured as throughput per FPGA slice, highlights an interesting trade off between simulation time and area used. To recall, the area used is a derived area that takes the total amount of Slice Flip Flops, LUTs, DSP48s, and BRAMs and then multiplies

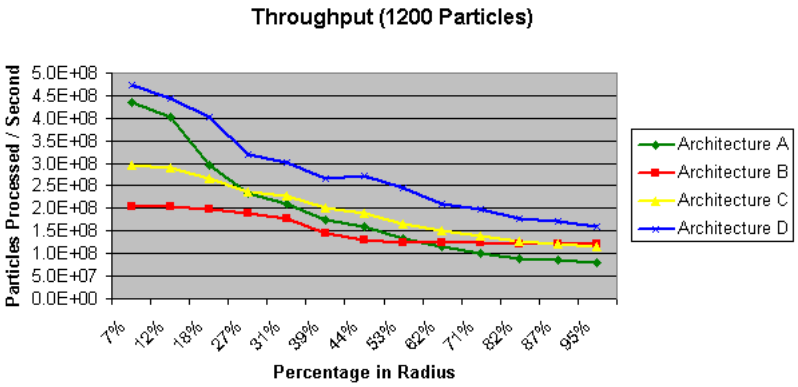

Fig. 5.6: Throughput for 1200 particles processed.

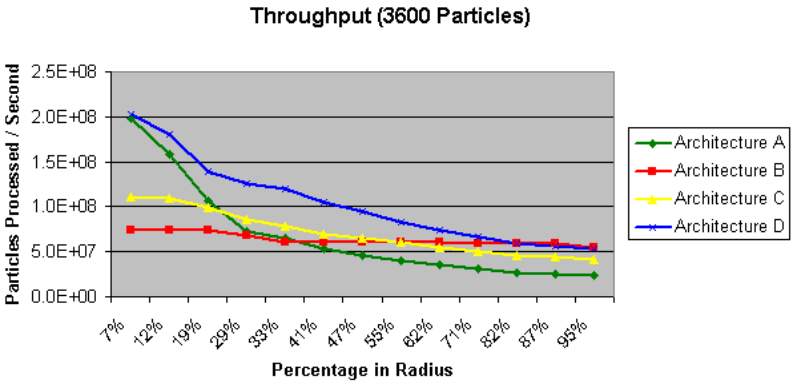

Fig. 5.7: Throughput for 3600 particles processed.

that number by the combined ratios of used out of available. The table that provides that information in table 5.2. Architecture A, being smallest in footprint, is ideal for simulations with few particles falling within. This small footprint renders high efficiency for small cutoff radii. That advantage is quickly lost, as is seen in figs. 5.8 and 5.9. Contrary to figures showing total number of clock cycles for particles processed (5.4 and 5.5), and throughput (5.6 and 5.7), the increasing behavior in the efficiency graphs (figs. 5.8 and 5.9) is desired. Architecture D, despite having a larger area due to the inclusion of the Flex unit, maintains its efficiency through its ability to process particles at a faster rate. Especially with larger cutoff radii, the efficiency achieved by Architecture D's architecture competes, and typically surpasses the three static architectures. These plots establish the contention that for certain circumstances, a static architecture may offer an overall better solution, the dynamic load

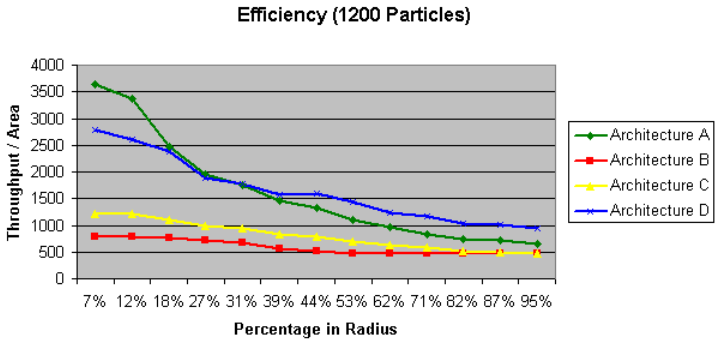

Fig. 5.8: Efficiency for 1200 particles processed.

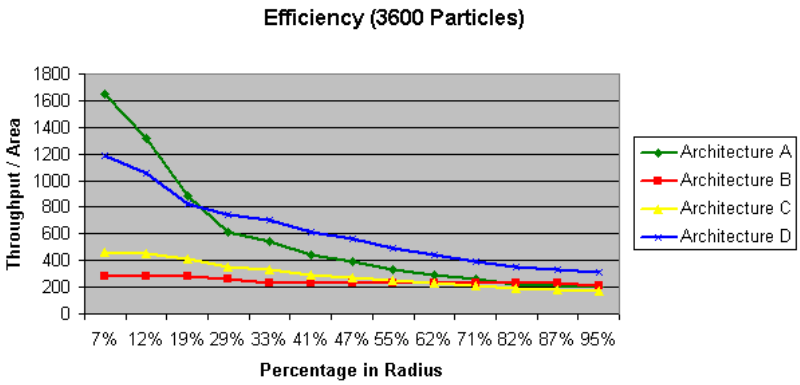

Fig. 5.9: Efficiency for 3600 particles processed.

balancing approach offers a more general solution that can be applied without modification to a variety of MD problems.

Chapter 6

Conclusions

When research on this topic began, the end goal of addressing load balancing was the first and final word on the subject. If the theory of using a dynamic, changing processing element can alleviate the architectural inefficiencies arising from unpredictability issues in Molecular Dynamics, then that increase in efficiency could potentially yield a number of technological advancements beneficial to society at large. During the course of the research, chance fell upon the work and there appeared a method to quickly develop alternate designs in order to further test the dynamic design. To conclude the research, each facet or goal of the project deserves attention and discussion.

6.1 Load Balancing

The issue of load balancing was addressed through the development of the Flex processor and supplemented with the architectural advantages offered by the FPGA. As was shown extensively in Chapter 5, the incorporation of the Flex unit on a dynamic architecture coupled with buffering and exploitation of spatial parallelism facilitated the address of dynamic load balancing. This impact was reflected by the input and output buffer levels emptying and filling at different rates dependent on how many particles were in radius at a given time. The plots that were reviewed are evidence that the Flex unit was switching at different times between different simulation steps due to the heaviest need.

Our methods for approaching load balancing allow the architecture to respond to dynamic changes in processing requirements. The ability to switch processing components on demand, together with the ability to direct data to available processing elements through intelligent buffering, facilitates an increase in processing efficiency. This point is further proven in seeing the throughput and efficiency of the Flex-based architecture match the

three static architectures for small, mid-range, and high cutoff radius values. The measures taken to address load balancing had a positive and measurable effect on throughput and efficiency. From the figures shown in Chapter 5, the Flex-based architecture had the best results overall for all cutoff radii for those two metrics.

6.2 Future Work

Improvements can be made in many areas, including but certainly not limited to the MD architecture itself, controller logic, processing element pipelining, and automatic scheduling. All of these measures would increase throughput and efficiency.

The MD architecture itself has room for improvement. From publications of related works it is seen that many researchers are interested in accelerating and using MD in a wide variety of applications; those specific to FPGA development were mentioned here. It is important to note that there are many ways to go about processing the data to obtain an MD simulation. It is established that the Flex unit greatly reduces wait time on a processing element level, but what was not explored was the various methods of implementing a Flex unit. It is possible that with the different FPGA fabrics out there and the ability to mix soft and hard core processors on the FPGA itself, future generations of FPGAs will be better suited to the control logic that is necessary to determine Flex modes. A soft core processor on an FPGA that is fabricated as such provides code caches and block rams that are specifically used for purposes such as the Flex unit. Migrating the Flex unit to that platform to explore the speedup of Flex processing times poses an interesting challenge and area of research. With the insanitation of many soft core processors, it is possible that an MD architecture could be made completely from Flex units that change from DC, to LJPC, to Verlet depending on the needs of the load.

Without completing the work to improve upon the design, it can be difficult to accurately predict the effects they will have on the results. It is natural to assume that once a given change is implemented, the desired effects are immediately rewarded. This is not always the case. However, it can be reasonably presumed that if an engineer were to address the most glaring deficiencies, then just a few tweaks or changes would greatly affect

the overall performance. For example, if one were to address the Flex and LJPC unit's large area, then it might be possible that more Flex units could fit on the chip and a few changes to the controller code would be all that is needed to integrate the new processing elements. A change of this scale would definitely lower total clock cycles and increase overall throughput and efficiency.

6.3 CHARGER Improvements

As was briefly mentioned in Chapter 5, an automatic code generator named CHARGER was attempted to quickly create comparative architectures and architecture simulators. The shortcomings of having hardware that performed similarly to the Flex-based architecture ultimately resulted in discontinuing its use in this project. There is opportunity to address the use of CHARGER for future endeavors. This would take work in several areas. One of these areas is the use of the automatically generated modules and the automatic scheduling of data to be processed. The usefulness of having an automatic hardware/software code generator can be stopped short if there is a need for human intervention when integrating the generated modules. Where this particular fix would start is in the generation of the state machine of each processing element generated by CHARGER. The naming convention for the parameters of each function is confusing. By keeping better track of data labeling, it is possible to produce a state machine that requires less human intervention for integration.

Customization of a project and time to develop a project make up an inverse relationship. Conceivably, with CHARGER, the time to develop could be shortened and that has a cost on customization. When the conditions are appropriate, and highly customized processing elements are not necessary, the engineer could use a well-developed CHARGER to his or her advantage.

6.4 Molecular Dynamics and Beyond

The issue of load balancing arises in other areas outside of Molecular Dynamics. Network traffic can be expressed in terms of an unpredictable load balance. Though the processes involved in network protocol resolution hardly compare to distance and force calcu-

lation, the principle of implementing a Flex-type processor to handle more than one type of task still applies. Use of such a Flex-processor in a network setting would increase resource efficiency.

Load balancing can also arise in terms of memory utilization. In this domain, there is no correlation between MD related processes (such as distance and force calculation) and memory utilization related processes. However, there still exists the issue of how best to use the demands of memory by the operating system. At any given time the user may request several applications and those applications may in turn request varying memory sizes. On the simplest of scales an embedded real-time operating system would highly benefit from a more efficient use of memory from a dynamic load balancing scheme. In such a case, the dynamic load balancing scheme would have to be able to fit on an embedded real-time platform, where memory reserves are typically small and processing power.

The development of this technology in an MD application such as the Blue Gene project could greatly enhance resources available to the machine running the MD application. As the Blue Gene project utilizes resources across a network of parallel machines, each node in the network could be treated as a processing element and be assigned either distance, force, inversion, Verlet, or a Flex processing scheme. Single processor implementations of MD, such as GROMACS, were they to add the elements of dynamic load balancing, would see a similar implementation as what was explored in this project as the platforms compare favorably on a macro-scale.

These changes and idea present interesting challenges for the future work of dynamic load balancing both within the scope of Molecular Dynamics and beyond.

References

[1] A. Nakano, R. K. Kalia, and P. Vashishta, "Scalable molecular-dynamics, visualization, and data-management algorithms for materials simulations," *Computing in Science and Engineering*, vol. 1, no. 5, pp. 39–47, 1999.

[2] C. G. Silva, V. Ostropytskyy, N. Loureiro-Ferreira, D. Berrar, M. Swain, W. Dubitzky, and R. M. M. Brito, "P-found: The protein folding and unfolding simulation repository," in *IEEE Symposium on Computational Intelligence and Bioinformatics and Computational Biology*, pp. 1–8. Los Alamitos, CA: IEEE Computer Society, 2006.

[3] T. Cagin, A. Jaramillo-Botero, G. Gao, and W. A. G. III, "Molecular mechanics and molecular dynamics analysis of drexler-merkle gears and neon pump," *Nanotechnology*, vol. 9, no. 3, pp. 143–152, 1998 [Online]. Available: `http://stacks.iop.org/0957-4484/9/143`.

[4] F. Allen, G. Almasi, and W. Andreoni, "Blue gene: A vision for protein science using a petaflop supercomputer," http://www.research.ibm.com/journal/sj/402/allen.html.

[5] G. Lienhart, A. Kugel, and R. Manner, "Using floating-point arithmetic on fpgas to accelerate scientific n-body simulations," in *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, p. 182. Los Alamitos, CA: IEEE Computer Society, 2002.

[6] L. Cordova and D. Buell, "An approach to scalable molecular dynamics simulation using supercomputing adaptive processing elements," in *International Conference on Field Programmable Logic and Applications*, pp. 711–712. Los Alamitos, CA: IEEE Computer Society, 2005.

[7] R. Scrofano and V. K. Prasanna, "Preliminary investigation of advanced electrostatics in molecular dynamics on reconfigurable computers," in *Supercomputing '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, p. 90. New York, NY: ACM Press, 2006.

[8] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow, "Reconfigurable molecular dynamics simulator," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pp. 197–206. Los Alamitos, CA: IEEE Computer Society, 2004.

[9] Y. Gu, T. VanCourt, and M. C. Herbordt, "Accelerating molecular dynamics simulations with configurable circuits," in *IEE Proceedings Computers and Digital Techniques*, pp. 189–195. Los Alamitos, CA: IEEE Computer Society, 2006.

[10] R. Scrofano and V. K. Prasanna, "Computing lennard-jones potentials and forces with reconfigurable hardware," in *Proceedings from International Conference on Engineering*

of Reconfigurable Systems and Algorithms, pp. 284–292. Los Alamitos, CA: IEEE Computer Society, 2004.

[11] C. Wolinski, F. Trouw, and M. Gokhale, "A preliminary study of molecular dynamics on reconfigurable computers," in *International Conference on Engineering Reconfigurable Systems and Algorithms*. Los Alamitos, CA: IEEE Computer Society, 2003.

[12] R. Scrofano, M. Gokhale, F. Trouw, and V. K. Prasanna, "Hardware/software approach to molecular dynamics on reconfigurable computers," in *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, pp. 23–34. Los Alamitos, CA: IEEE Computer Society, 2006.

[13] K. Sano, T. Iizuka, and S. Yamamoto, "Systolic architecture for computational fluid dynamics on fpgas," in *Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 107–116. Los Alamitos, CA: IEEE Computer Society, 2007.

[14] Y. Gu, T. VanCourt, D. DiSabello, and M. C. Herbordt, "Preliminary report: Fpga acceleration of molecular dynamics computations," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pp. 269–270. Los Alamitos, CA: IEEE Computer Society, 2005.

[15] GROMACS, "Gromacs md simulator," http://www.gromacs.org.

[16] R. Hayashi and S. Horiguchi, "Efficiency of dynamic load balancing based on permanent cells for parallel molecular dynamics simulation," in *International Conference on Parallel and Distributed Processing Symposium*, p. 85. Los Alamitos, CA: IEEE Computer Society, 2000.

[17] R. Hayashi and S. Horiguchi, "Relationships between efficiency of dynamic load balancing and particle concentration for parallel molecular dynamics simulation," in *Proceedings of the The Fourth International Conference on High-Performance Computing in the Asia-Pacific Region-Vol. 2*, p. 976. Washington, DC: IEEE Computer Society, 2000.

[18] A. D. Serio and M. B. Ibez, "Distributed load balancing for molecular dynamics simulations," in *Annual International Symposium on High Performance Computing Systems and Applications*, p. 283. Los Alamitos, CA: IEEE Computer Society, 2002.

[19] A. Nakano, R. K. Kalia, and P. Vashishta, "Scalable molecular-dynamics, visualization, and data-management algorithms for materials simulations," *Computing in Science and Engineering*, vol. 1, no. 5, pp. 39–47, 1999.

[20] J. S. Jean, K. Tomko, V. Yavagal, J. Shah, and R. Cook, "Dynamic reconfiguration to support concurrent applications," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 591–602, 1999.

[21] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 4, pp. 289–299, 2005.

[22] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "On the interest of load balancing asynchronous parallel iterative algorithms," Technical Report -LIFC.

[23] R. Durikovic and T. Motooka, "Molecular dynamics simulation and visualization," in *Third International Conference on Information Visualisation (IV'99)*, p. 334. Los Alamitos, CA: IEEE Computer Society, 1999.

[24] Y. Gu, T. VanCourt, and M. Herbordt, "Improved interpolation and system integration for fpga-based molecular dynamics simulations," in *International Conference on Field Programmable Logic and Applications*, pp. 1–8. Los Alamitos, CA: IEEE Computer Society, 2006.

[25] Y. Gu and M. C. Herbordt, "Fpga-based multigrid computation for molecular dynamics simulations," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 23–25. Los Alamitos, CA: IEEE Computer Society, 2007.

[26] G. Danese, I. de Lotto, F. Leporati, and A. Spelgatti, "Fpga based coprocessor to calculate the energy of dipolar system," in *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pp. 20–27. Los Alamitos, CA: IEEE Computer Society, 2002.

[27] M. Choi and N. Park, "Dynamic yield analysis and enhancement of fpga reconfigurable memory systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 51, no. 6, pp. 1300–1311, 2002.

[28] V. Dandalis, A.; Prasanna, "Configuration compression for fpga-based embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 12, pp. 1394–1398, 2005.

[29] K. Eguro and S. Hauck, "Resource allocation for coarse-grain fpga development," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1572–1581, 2005.

[30] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, 2004.

[31] Y. Yu, A. Srinivasan, and N. Chandra, "Scalable time-parallelization of molecular dynamics simulations in nano mechanics," in *International Conference on Parallel Processing*, pp. 119–126. Los Alamitos, CA: IEEE Computer Society, 2006.

[32] K. J. Bowers, E. Chow, H. Xu, R. O. Dror, M. P. Eastwood, B. A. Gregersen, J. L. Klepeis, I. Kolossvary, M. A. Moraes, F. D. Sacerdoti, J. K. Salmon, Y. Shan, and D. E. Shaw, in *Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters*, p. 43. Los Alamitos, CA: IEEE Computer Society, 2006.

[33] J. Emmert, C. Stroud, and M. Abramovici, "Online fault tolerance for fpga logic blocks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 2, pp. 216–226, 2007.

[34] I. Robertson and J. Irvine, "A design flow for partially reconfigurable hardware," *Transactions on Embedded Computing Systems*, vol. 3, no. 2, pp. 257–283, 2004.

[35] L. Shang and N. K. Jha, "Hardware-software co-synthesis of low power real-time distributed embedded systems with dynamically reconfigurable fpgas," in *International Conference on VLSI Design*, p. 283. Los Alamitos, CA: IEEE Computer Society, 2002.

[36] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. DiNola, and J. R. Haak, "Molecular dynamics with coupling to an external bath," *The Journal of Chemical Physics*, vol. 81, no. 8, pp. 3684–3690, 1984 [Online]. Available: `http://link.aip.org/link/?JCP/81/3684/1`.

[37] J. Phillips, M. Areno, C. Rogers, A. Dasu, and B. Eames, "A reconfigurable load balancing architecture for molecular dynamics," in *International Symposium on Parallel and Distributed Processing*, p. 187. Los Alamitos, CA: IEEE Computer Society, 2007.