

A Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers *

Ronald Scrofano[†], Maya Gokhale[‡], Frans Trouw[‡], and Viktor K. Prasanna[†]

[†]University of Southern California, Los Angeles, CA

[‡]Los Alamos National Laboratory, Los Alamos, NM

{rscrofan, prasanna}@usc.edu, {maya, trouw}@lanl.gov

Abstract

With advances in reconfigurable hardware, especially field-programmable gate arrays (FPGAs), it has become possible to use reconfigurable hardware to accelerate complex applications, such as those in scientific computing. There has been a resulting development of reconfigurable computers—computers which have both general purpose processors and reconfigurable hardware, as well as memory and high-performance interconnection networks. In this paper, we study the acceleration of molecular dynamics simulations using reconfigurable computers. We describe how we partition the application between software and hardware and then model the performance of several alternatives for the task mapped to hardware. We describe an implementation of one of these alternatives on a reconfigurable computer and demonstrate that for two real-world simulations, it achieves a $2\times$ speed-up over the software baseline. We then compare our design and results to those of prior efforts and explain the advantages of the hardware/software approach, including flexibility.

1 Introduction

Reconfigurable hardware, in the form of FPGAs, has been used successfully in the acceleration of many applications and application tasks. Previously, FPGAs were limited to performing tasks that required only fixed-point arithmetic. However, recent advances in FPGA hardware, including increased density and the inclusion of embedded multipliers, have made floating-point arithmetic on FPGAs possible. Several libraries for performing floating-point arithmetic on FPGAs have been developed [4, 5, 6]. An exciting challenge is to use FPGAs to accelerate scientific

computing applications that are very computationally intensive and require floating-point arithmetic.

One particularly interesting scientific computing application to investigate is molecular dynamics (MD) simulation. MD is a technique that models the movements of atoms in a substance over time. Tasks in the simulation include calculating forces, updating positions and velocities, and other supporting tasks. All of these calculations are traditionally performed in single- or double-precision floating-point arithmetic. One task in an MD simulation—the nonbonded force calculation—is very computationally intensive while the other tasks are less intensive or have complicated control logic. It makes sense, then, to accelerate the nonbonded force calculation in reconfigurable hardware while executing the rest of the simulation with a general purpose processor.

In this paper, we describe designing and implementing an MD simulation system using such a hardware/software approach. The implementation platform is a reconfigurable computer that has both general purpose processors and reconfigurable hardware. We first develop a software implementation that has most of the important tasks in an MD simulation but is not as complex as large MD software packages such as GROMACS and NAMD [8, 16]. We then partition the tasks in the simulation such that the most intensive is executed in reconfigurable hardware and the rest are executed by the general purpose processor.

This hardware/software approach to acceleration is the key to developing a more complete simulation system than previous acceleration efforts that moved all tasks of the simulation into hardware [3, 7]. It takes into account the fact that not all tasks in a large scientific program, such as a program for MD simulation, are good candidates for acceleration. Tasks which are control-intensive or which can be executed very efficiently by general purpose processors should be left in software. Acceleration efforts should be targeted towards those tasks whose acceleration will have the greatest impact on the performance of the overall al-

*This work is supported by Los Alamos National Laboratory under contract/award number 95976-001-04 3C.

gorithm. Another advantage of the hardware/software approach is that it allows for incremental acceleration of the application. That is, once even a single task is implemented in hardware, the application is accelerated. Other tasks may be accelerated later to provide even greater speed-ups, but at no time is functionality lost. In a hardware-only approach, the entire application must be accelerated before any benefit is realized. Similarly, the hardware/software approach gives the system the flexibility to easily integrate new tasks for advanced simulations. These can first be integrated as software, then, if necessary, accelerated with hardware.

In the next subsection, we describe reconfigurable computers. In Section 2, we introduce MD simulations and describe our software. In Section 3, we describe the hardware/software partitioning. We then model several alternatives for the reconfigurable hardware implementation and decide, based upon estimated performance of the whole application, which to implement. In Section 4, we describe our implementation on a reconfigurable computer and compare its performance to that of the pure software approach and to that predicted in Section 3. We discuss related work and compare it to ours in Section 5. Finally, in Section 6, we draw some conclusions and discuss future directions.

1.1 Reconfigurable Computers

Reconfigurable computers are comprised of both general purpose processors and FPGAs, as well as high-performance, low-latency interconnect between them. The FPGAs act as programmable application accelerators for the general purpose processors. Reconfigurable computers are ideal for executing applications that contain both control-intensive portions, which execute on the general purpose processors, and data-intensive portions, which execute on the FPGAs. While in this work we use a single general purpose processor and a single FPGA, the high-performance nature of reconfigurable computers makes them well-suited for use in large clusters and supercomputers.

In this paper, we will consider one specific reconfigurable computer: the SRC 6e MAPstation [15]. The SRC 6e MAPstation has two 2.8 GHz Intel Xeon processors; a *MAP processor*, which has two Xilinx Virtex-II FPGAs available for custom designs; and a high-performance interface connecting them. The choice of which portions of the application execute on which processing devices is left up to the designer. The development environment supports the design of accelerators in C, Fortran, VHDL, and Verilog.

2 Molecular Dynamics

MD is a widely used technique for simulating the movements of atoms in a system over time. The number of atoms in a system varies widely, from about 10000 atoms in small

simulations to over a billion atoms in the largest simulations. In this section, we provide background information about MD, the reference for which is [1].

In an MD simulation, each atom i in the system has an initial position $\vec{r}_i(0)$ and velocity $\vec{v}_i(0)$ at time $t = 0$. Given the initial properties of the system, the MD simulation determines the *trajectory* of the system from time $t = 0$ to some later time $t = t_f$. In the process, the simulation keeps track of properties such as temperature and total energy.

In order to compute the system’s trajectory, the positions of all the atoms at time $(t + \Delta t)$ are calculated based on the positions of all the atoms at time t , where Δt is a small time interval, typically on the order of one femtosecond. One of the most popular methods for calculating new positions is the velocity Verlet algorithm. The steps in this algorithm are, for each atom i

1. calculate $v_i(t + \frac{\Delta t}{2})$ based on $v_i(t)$ and acceleration $a_i(t)$;
2. calculate $r_i(t + \Delta t)$ based on $v_i(t + \frac{\Delta t}{2})$;
3. calculate $a_i(t + \Delta t)$ based on $r_i(t + \Delta t)$ and $r_j(t + \Delta t)$ for all atoms $j \neq i$;
4. calculate $v_i(t + \Delta t)$ based on $a_i(t + \Delta t)$.

The most time-consuming part of the simulation is updating the acceleration. The acceleration comes from the forces acting upon each of the atoms.

2.1 Force Calculation

The forces in an MD simulation can be grouped in two categories: bonded and nonbonded. Bonded forces—namely bond stretch, angle bend, and dihedral torsion—only occur between atoms that are bonded to one another. Thus, each atom only interacts with a few other atoms and the computation of bonded forces is not a bottleneck.

Nonbonded forces, on the other hand, can occur between any two atoms in a simulation that are not part of the same bond. The two types of nonbonded forces most often used in MD simulations are the forces due to the Lennard-Jones and Coulomb potentials (hereafter referred to as the Lennard-Jones force and the Coulomb force, respectively).

Equation 1 must be evaluated for every atom i in a simulation to find the nonbonded force acting on atom i . The first two terms of the equation make up the Lennard-Jones force and the third makes up the Coulomb force. \vec{r}_{ij} is the distance vector between atoms i and j and r_{ij} is the magnitude of \vec{r}_{ij} , i.e., the distance between atoms i and j . A and B are constants that depend on the types of atoms i and j . q_i is the charge on atom i .

$$\vec{f}_i = \sum_{j \neq i} \left(\frac{12A}{r_{ij}^{14}} - \frac{6B}{r_{ij}^8} + \frac{q_i q_j}{4\pi \epsilon_0 r_{ij}^3} \right) \vec{r}_{ij} \quad (1)$$

In practice, it is common to use Newton’s Third Law, namely that $\vec{f}_{ij} = -\vec{f}_{ji}$, to reduce the number of calculations. So for each pairwise force calculated, \vec{f}_{ij} is added to \vec{f}_i and subtracted from \vec{f}_j .

When calculating the Lennard-Jones forces, a *cutoff distance* is always employed. That is, f_{ij}^{LJ} is only calculated if r_{ij} is less than some distance r_c . There are several methods for calculating the Coulomb force, including cutoff Coulomb, which uses the same cutoff technique as is used for Lennard-Jones forces, Ewald summation, and Particle Mesh Ewald (PME).

One of the difficulties of nonbonded force calculation is that each pair of atoms may interact. Naively, this would require $O(n^2)$ evaluations at each step, as the distance between each atom and every other atom must be checked. Two techniques to reduce the number of pairs that must be evaluated are the *linked cell list* and the *Verlet neighbor list*.

In the linked cell list approach, the simulation box is broken into a number of cells, where the length, width, and height of each cell is close to but greater than the cutoff distance. The atoms in each cell, then, only interact with atoms in the same cell or the 26 neighboring cells. This drastically reduces the number of pairs of atoms that must be evaluated.

The purpose of the neighbor list is to avoid searching for interacting pairs of atoms at every time step. It is based on the principle that if atoms i and j are close at time step t , they should still be close at time step $t + 1$. So, the neighbor list is built at time step t and updated at time step $t + x$, where x is commonly between 10 and 20. The neighbor list is really a combination of lists, where each atom has a list of atoms with which it might interact. Throughout the paper, when we refer to “the neighbor list,” we mean the all-encompassing combination of lists. When we refer to “atom i ’s neighbor list,” we are referring to only the list of atom i ’s neighbors. The list can be constructed using either the $O(n^2)$ search for pairs or the more efficient linked cell list approach.

The first step in implementing the MD simulation application on a reconfigurable computer is to develop a software implementation that can be profiled and eventually modified to make use of the reconfigurable hardware.

2.2 Software Implementation

For our work, we have written a software implementation of a molecular dynamics simulation. This software implements the Velocity Verlet algorithm in single or double precision arithmetic. For nonbonded force calculation, we calculate the Lennard-Jones force and use the cutoff Coulomb technique for the Coulomb force. During force calculation, we also calculate the potential energy. For better accuracy, we use shifted forces and potential energies to remove discontinuities in the force and potential energy at

the cutoff distance [1]. Doing so adds the need for two more type-based constants and several extra floating-point operations to the force calculation. To identify the interacting pairs of atoms, we use the Verlet neighbor list technique. Every x steps, where x is defined by the user, a linked cell list is created. This linked cell list is then traversed to make the neighbor list. We have also implemented the bonded force calculation, referencing [2] while doing so.

The inputs to the simulation are provided at run-time. They include the number of steps in the simulation, the length of the time step, the dimensions of the simulation box, the cutoff and neighbor list cutoff distances, and the frequency at which the neighbor list should be updated. An Amber-format topology file can be used to specify the structure of the system [2]. This specification includes the types, charges, and masses of the atoms, the groups of atoms that participate in bonded interactions, and constants used in both bonded and nonbonded force calculation. The initial positions of the atoms in the system can be specified in a coordinate file in PDB format [11].

The software is written in C. It has been compiled using Intel’s C compiler, version 8.1. We profiled the single-precision version of the software running two benchmark simulations for 1000 steps each. We ran the simulations on one of the 2.8 GHz Xeon processors in the MAPstation. The operating system is Linux. To profile, we used Oprofile version 0.8.1 [9]. Table 1 shows the profiling results.

2.2.1 Notes on the Implementation

The simulations described in this paper follow conventional methodologies as used in the simulation community. There are two approximations made. One is the use of single-precision floating-point arithmetic. While our purely software version can use single- or double-precision, the task running in reconfigurable hardware is limited to single-precision, as will become evident in the following section.

The use of double-precision arithmetic is the conventional approach in MD simulations, although the widely used GROMACS software is often used with single-precision arithmetic [16]. The original driver for using high precision arithmetic was the desire to avoid cumulative rounding errors implicit in very long simulations, but this has been of less concern in recent work. Unless divergence from the “exact” atomic trajectories is important, a system using single-precision arithmetic will still explore realistic configurations during the simulation [12].

The other approximation is the cutoff Coulomb approach for the Coulomb forces. This is more of a concern than using single- instead of double-precision arithmetic. Recent simulations that compare the structure of biologically inspired model membrane structures have demonstrated that the bilayer structures of cells are different at approx-

Table 1. Profile of the software implementation for two benchmark simulations

Task	% of Computation Time	
	Palmitic Acid	CheY Protein
Nonbonded Forces	75.15	74.09
Building Neighbor List	20.75	22.76
Dihedral Torsion Forces	2.14	1.51
Other	1.96	1.64

imately the 5% level compared to when more sophisticated approaches such as PME are used [10]. There are also indications that the cutoff Coulomb approach creates incorrect structures for simulations of solvated proteins and DNA. However, this effect is still only at the 5% level or less, and for some simulations the computational efficiency inherent in this approach makes it the only realistic choice.

2.3 Benchmark Simulations

As benchmarks, we use simulations of palmitic acid and the CheY protein. Palmitic acid self-assembles into a one-molecule-thick film on water. There is very detailed experimental information that describes the atomic level structure of this film, and we are simulating such monolayers to see whether the current parameterization of the interatomic interactions used in simulations can reproduce this behavior.

In this simulation, there are 52558 atoms of 8 different types. The timestep for the simulation is 2 fs. The cutoff distance is 10 Å and the neighbor list cutoff distance is 12 Å. The neighbor list is built once every 10 steps. The simulation box is $104 \times 84.8 \times 256 \text{ \AA}^3$.

The interest in simulating the dynamics of the CheY protein is the surprising observation that microbial life forms can exist in very hot environments. There are many variants of this protein and we are simulating two variants to come to an understanding of their differing thermal stabilities.

In this simulation, there are 32932 atoms of 17 different types. As is the case for palmitic acid, the timestep is 2 fs, the cutoff distance is 10 Å, the neighbor list cutoff distance is 12 Å, and the neighbor list is built once every 10 steps. The simulation box is $73.8 \times 71.8 \times 76.8 \text{ \AA}^3$.

3 Design Evaluation

From the profile in Table 1, it is clear that accelerating the computation of nonbonded forces would benefit the overall application most. But, before implementing the nonbonded force calculation in reconfigurable hardware, we need to determine if doing so will provide a significant speed-up for the overall application. To make this determi-

nation, we model the performance of reconfigurable computer implementations before implementing them.

We model the reconfigurable computer as consisting of two types of nodes: *general-purpose-processor (GPP) nodes* and *reconfigurable-hardware (RH) nodes*. GPP nodes consist of one or more general purpose processors and memory. RH nodes consist of reconfigurable hardware, likely in the form of one or more FPGAs, and on-board memory banks. Tasks in the application execute on either a GPP node or an RH node, but not both. In order for a task to execute on a particular node, the data needed by the task must reside in the node’s memory. If it does not, it must be transferred there, incurring a communication cost.

In this paper, we are using one general purpose processor on a single GPP node and one FPGA on a single RH node. Further, we are only moving one task to the RH node. Let T_{SWtasks} be the time taken by the tasks executing in software, T_{comm} be the time required to transfer the necessary data, and T_{RH} be the time taken by the nonbonded force calculation executing in reconfigurable hardware. The total time required to perform a step of the MD simulation on a reconfigurable computer is given by the sum of T_{SWtasks} , T_{comm} , and T_{RH} .

Finding T_{SWtasks} is trivial: we simply use the profiling data from the software-only version of the application. Finding T_{comm} is also fairly simple: we determine how much data is transferred from/to GPP node to/from RH node and then divide by the bandwidth available between the two nodes. Determining T_{RH} is more challenging.

T_{RH} is affected most by the parallelism and pipelining of the design. Parallelism is limited by the available logic resources of the reconfigurable hardware and the per-cycle bandwidth between the reconfigurable hardware and its on-board memory. Data dependencies and hazards involved with accessing local memory can cause pipeline stalls that are detrimental to the overall performance.

Taking these factors into account, it is possible to estimate the number of cycles a design will require to produce a result. Estimates for achievable frequency can then be used to determine the execution time.

3.1 System-Level Design

At the system level, the partitioning of the MD simulation algorithm between hardware and software is as shown in Figure 1. At each time step, the current atomic positions need to be transferred to the RH node’s on-board memory. Each position vector has three components. So, if n is the number of atoms and b is the number of bytes per floating-point word, $3nb$ bytes must be transferred to the RH node’s on-board memory. Further, the neighbor list must be transferred to the RH node, although it should be possible to stream it to the RH node, thus overlapping the communica-

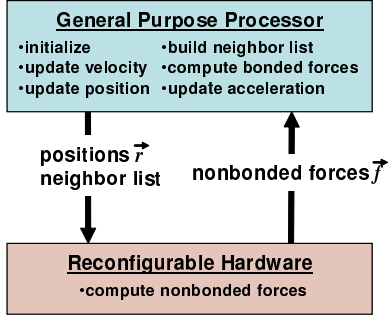


Figure 1. Hardware/software partitioning of the MD simulation application

tion with computation. The reconfigurable hardware must also have access to the types and charges of each atom, which are constant throughout the simulation but vary from simulation to simulation. Several other values that are constant throughout the simulation must be available as well, such as the size of the simulation box and the constants in the Lennard-Jones force equations.

As the nonbonded forces are calculated, they will be stored in the RH node’s on-board memory. The force acting on each atom has three components, so the amount of data that needs to be stored is $3nb$. At the end of the nonbonded force calculation, this data will be transferred back to the GPP node. Therefore, ignoring the time taken to transfer data that remains constant since it is only transferred once and ignoring the time to transfer the neighbor list since we assume that it will be streamed, the total communication time, T_{comm} , is $6nb/B_{GPP,RH}$ s/step, where $B_{GPP,RH}$ is the bandwidth between the GPP node and the RH node.

3.2 RH Node-Level Design Alternatives

The basic design for performing the nonbonded force calculation task on the RH node is a direct translation of software into hardware. The algorithm and architecture for such a design are shown in Figures 2 and 3, respectively. Though omitted in the figures, the type and charge of each atom is also used in the calculation. In the figures, `positionOBM` and `forceOBM` represent on-board memories. In Figure 2, `CALC_NBF` represents applying the shifted force version of Equation 1 and finding the potential energy, as well as other necessary techniques, such as the *minimum image convention* [1].

In this design, the two loops in the algorithm are pipelined such that in every cycle, either a new i atom is entering the pipeline or a new j atom is entering the pipeline. If $r_{ij} \geq r_c$, the results of the calculation are discarded at the end of the pipeline. In Figure 3, the “force calculation” box represents this pipeline. One problem with this design is that in order to subtract a newly calculated \vec{f}_{ij} from \vec{f}_j , \vec{f}_j needs to be *read* from the force memory. If, as is common in

```

1 foreach atom  $i$  do
2    $\vec{r}_i \leftarrow \text{positionOBM}[i]$ 
3    $\vec{f}_i \leftarrow \text{forceOBM}[i]$ 
4   foreach neighbor  $j$  of  $i$  do
5      $\vec{r}_j \leftarrow \text{positionOBM}[j]$ 
6     if  $|\vec{r}_i - \vec{r}_j| < r_c$  then
7        $\vec{f}_{ij} \leftarrow \text{CALC\_NBF}(\vec{r}_i, \vec{r}_j)$ 
8        $\vec{f}_i \leftarrow \vec{f}_i + \vec{f}_{ij}$ 
9        $\vec{f}_j \leftarrow \text{forceOBM}[j]$ 
10       $\text{forceOBM}[j] \leftarrow \vec{f}_j - \vec{f}_{ij}$ 
11    end
12  end
13   $\text{forceOBM}[i] \leftarrow \vec{f}_i$ 
14 end

```

Figure 2. Algorithm for basic design

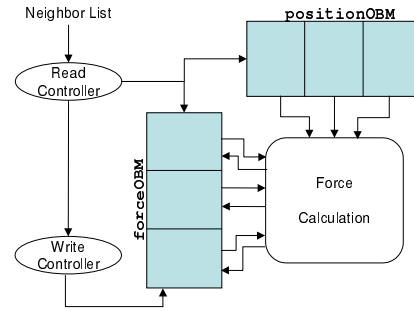


Figure 3. Architecture for basic design

presently available reconfigurable computers, the on-board memory on the RH node is single ported, the pipelined architecture will have to stall for one cycle. There may also be penalties associated with switching the memory between read mode and write mode.

Because of the lengthy pipelines in floating-point cores, this design is also susceptible to a read-after-write hazard. Specifically, the subtraction in line 10 of the algorithm may still be taking place for one i atom when another i atom causes the read in line 9 to execute. Since it is not possible to forward values within the floating-point cores, stalling will be necessary.

The time required to do the calculations with this design, in cycles, is given in Equation 2. L is the length of the neighbor list and s is average the number cycles of stalling due to conflicting memory accesses and hazards. Put another way, on average, the pipeline produces one new result every s cycles.

$$T_{RH} = sL \quad (2)$$

A second alternative is to use a *write-back design*. The algorithm for this design is given in Figure 4. This design makes use of the fact that an atom j occurs in atom i ’s neighbor list at most once. The loop beginning on line 5 is

pipelined, as is the loop in beginning on line 16. The outer loop is not pipelined. The key is that the new \vec{f}_j values are not stored directly into on-board memory but instead are stored temporarily in on-chip memory (forceRAM). They are not written back to on-board memory until all of the neighbors of the current i atom have been processed. Because force calculation for a new i atom cannot begin until after all of the j atoms from the previous i atom's neighbor list have been written back to memory, the read-after-write hazard is avoided. The conflicting memory accesses of the basic design are also avoided. The number of times that the on-board memory switches modes is also minimized. The drawback to this design is that it requires that the pipeline be drained after the processing of each i atom. Thus, the pipeline fill-up latency is incurred for each i atom.

To lessen the number of pipeline stages that must be drained after each i atom, the pipelined inner loop can be broken into two separate pipelines joined by a FIFO. This FIFO must be large enough to hold all of the neighbors of two (or more) consecutive i atoms. The first pipeline reads the position, charge, and type data for each neighbor atom j and calculates \vec{r}_{ij} . If $r_{ij}^2 < r_c^2$, \vec{r}_{ij} , the charge of j , and the type of j are written to the FIFO. Note that there are no dependencies at this stage, so this pipeline can write new values to the FIFO as long as there is space. When the FIFO is full, it waits until the second pipeline has processed data before it starts writing new values. The second pipeline reads data from the FIFO and does the remaining steps in the force calculation. This pipeline must be drained after each new i atom that it reads from the FIFO. Since the first pipeline should produce data faster than the second pipeline can process it, there should be data ready in the FIFO whenever the second pipeline is ready to begin processing it. Effectively, then, it is only the second pipeline that must be drained and whose fill-up latency is incurred for each atom.

The time required to do the calculations with this design, in cycles, is given in Equation 3, where n is the number of atoms, N is the average number of neighbors for each atom, p is the number of pipeline stages (if two pipelines are used, p can be considered the number of stages in the second pipeline), and s is the number of cycles required to switch an on-board memory from read mode to write mode and vice versa. The first term in Equation 3 gives the number of cycles spent in the (second) pipeline. The second term gives the number of cycles spent writing data back to memory, assuming these writes can be pipelined. The third term gives the number of cycles spent reading i atoms' data from memory. The last term gives the number of cycles spent switching the on-board memory between read and write modes.

$$T_{RH} = n(N + p) + n(N + 1) + n + 2sn \quad (3)$$

The last design alternative does not use that $\vec{f}_{ij} = -\vec{f}_{ji}$. The algorithm is the same as that in Figure 2, except without

```

1 foreach atom  $i$  do
2    $\vec{r}_i \leftarrow \text{positionOBM}[i]$ 
3    $\vec{f}_i \leftarrow \text{forceOBM}[i]$ 
4    $n \leftarrow 0$ 
5   foreach neighbor  $j$  of  $i$  do
6     if  $|\vec{r}_i - \vec{r}_j| < r_c$  then
7        $\vec{r}_j \leftarrow \text{positionOBM}[j]$ 
8        $\vec{f}_{ij} \leftarrow \text{CALC\_NBF}(\vec{r}_i, \vec{r}_j)$ 
9        $\vec{f}_i \leftarrow \vec{f}_i + \vec{f}_{ij}$ 
10       $\vec{f}_j \leftarrow \text{forceOBM}[j]$ 
11      forceRAM[ $n$ ]  $\leftarrow \vec{f}_j - \vec{f}_{ij}$ 
12       $n \leftarrow n + 1$ 
13    end
14  end
15  forceOBM[ $i$ ]  $\leftarrow \vec{f}_i$ 
16  foreach  $\vec{f}_j$  in forceRAM do
17    forceOBM[ $j$ ]  $\leftarrow \vec{f}_j$ 
18  end
19 end

```

Figure 4. Algorithm for write-back design

lines 9 and 10. This design requires twice as many force calculations but removes the hazards and the need to read from the force memory. A caveat is that using this method will require use of a second neighbor list: if j is a neighbor of i in the original list, then i must be a neighbor of j in the second neighbor list. The time required to do the calculations with this design, in cycles, is given in Equation 4, where $T_{\text{BNL,GPP}}$ is the amount of time it takes to build the extra neighbor list on the general purpose processor. While this latter time is incurred by a task taking place on the general purpose processor, we include it here to emphasize that this method requires building the second neighbor list.

$$T_{RH} = 2L + T_{\text{BNL,GPP}} \quad (4)$$

Note that Equations 2, 3, and 4, assume that there is enough bandwidth between the reconfigurable hardware and its on-board memory to read all three components of the positions, as well as the types and charges in parallel, while writing all three components of the forces in parallel.

3.3 Choosing a Design for the Implementation

We now use Equations 2, 3, and 4 to determine which design we should use in our reconfigurable computer implementation. We are only able to do the nonbonded force computation in single-precision arithmetic on the reconfigurable hardware. Using double precision arithmetic would require more logic than is available in the FPGAs in the MAPstation. Even if the amount of logic were sufficient,

there would not be enough on-board memory to store all of the necessary data. The MAPstation has eight banks, each of which is eight bytes wide, which is not enough to hold position, force, charge, and type data and still have a bank available for streaming the neighbor list. Thus, we focus on single-precision arithmetic and b , the number of bytes in a floating-point word, is four.

An important property of the MAPstation is its requirement that all hardware designs run at 100 MHz. While this does not guarantee that any of the above designs will run at 100 MHz, 100 MHz is the best possible frequency, so we use it in estimating times. Two other important properties are that the on-board memory banks are single ported and that switching a memory from write mode to read mode and vice versa requires four dead cycles [15].

We use the timing information from our palmitic acid simulation to make our decisions. From profiling, we have found that on average, a single step of the velocity Verlet algorithm takes 1.27 s to execute in software. $T_{S\text{Wtasks}}$ is 0.32 s/step and will be the same for each of the architectures.

T_{comm} will also be the same no matter which architecture is chosen. We want to stripe the 3 4-byte position components over 2 memory banks (each 8 bytes wide) so that all 3 can be accessed in parallel. In order to achieve this, we must transfer 4 words for each set of components: the 3 position components and 1 extra word. A similar situation arises in writing the forces back to the GPP node. So, rather than transferring $6n$ words, we actually transfer $8n$ words per step. With the 52558 atoms in the palmitic acid simulation, this translates to transferring 1.7 MB per step at the MAPstation's transfer rate of 1400 MB/s, requiring only about 1.2 ms per step [15].

The basic design produces poor results. Because of the memory dependencies, the pipeline will only produce a result once every 10 cycles. The neighbor list in the palmitic acid simulation has about 17 million entries, so the number of cycles required per step is 170 million. This leads to an overall time of about 2 s/step, which is a slowdown over the software only version.

The write-back design fares better. From our knowledge of the floating-point cores and the structure of the force calculation pipeline, we know that the pipeline will be very long, about 200 stages. Through profiling the simulation, we know that the average number of neighbors close enough to interact is 192. With this and other information about the system, we find that even with the long pipeline, the anticipated total time is 0.63 s/step, a $2\times$ speedup over the software implementation.

The third design is hampered by the need to build a second neighbor list and gives a speedup that is lower than that of the write-back approach. The time spent in force computation is only 0.34 s/step, but every 10 steps, the neighbor list is rebuilt. Building the second neighbor list contributes

an average of 0.27 s/step extra. The total is then 0.93 s/step, only a $1.4\times$ speed-up. Clearly, the alternative to use is the write-back approach.

4 Implementation and Experimental Results

For implementation on the MAPstation, we made a few modifications to the traditional techniques. The neighbor list is usually implemented by a long list of atoms with a separate array that points to the start of a particular atom's set of neighbors. To reduce the number of separate arrays, we instead insert the atoms into the neighbor list right before their respective sets of neighbors. We use the most significant bit of the word as a flag to denote which atoms start new sets. To save memory banks in the RH node, we do not store the atom type array in its own on-board memory bank. Instead, we pack the type and the atom into the neighbor list in one 64-bit word. This does not adversely affect the communication time because we are streaming in the neighbor list in 64-bit words anyway. We pack the charge in the same array as the positions because, as mentioned above, we must transfer 4 4-byte words, instead of 3, every step anyway.

The first time the nonbonded force calculation is called, the constants for the simulation are transferred to the FPGA. The dimensions of the simulation box are stored in registers. The constants for Lennard-Jones force calculation are stored in four block RAMs, one each for the A , B , and the two force/potential shift constants.

Each time the nonbonded force calculation task is executed, the position and charge data is transferred before calculation starts. Due to limitations with the MAPstation's streaming DMA, we are unable to stream the neighbor list. Instead, we use regular DMAs but overlap them with computation. We first DMA a section of the neighbor list into two on-board memory banks. While the force calculation is processing that section of the neighbor list, we DMA the next section of the neighbor list into another two on-board memory banks. We switch back and forth between reading and writing memory banks so that the computation is overlapped with communication.

For intermediate storage, we use block RAMs. From profiling our simulations, we found that no atom had more than 750 neighbors. The block RAMs in the target Xilinx FPGA hold 512 32-bit words each. Thus, for the FIFO between the distance- and force calculation pipelines, we use four block RAMs each for the the three components of the distance vector, the squared magnitude of the distance vector, the type of the atom, and the charge of the atom. Thus, we use 24 block RAMs to implement the FIFO. In the force calculation pipeline, we use two block RAMs per force component as intermediate storage. We use another two block RAMs to store the index values of the neighbors.

When the forces are written back to on-board memory,

they are striped across two on-board memories. Upon completion of the nonbonded force calculation, the forces are DMAed back to the general purpose processor.

The code is written in C for the SRC Carte compiler. The floating-point cores provided by SRC are used to carry out the floating-point arithmetic. To accumulate the force acting on an i atom, we use three of the accumulator floating-point cores provided by SRC, one for each component of the force. We use another accumulator core to accumulate the potential energy. This floating-point core allows one new input to be presented to the accumulator per clock cycle. When a new accumulation is to start (in our case, when force calculation for a new i atom begins), the accumulator pipeline must be flushed. This flushing of the accumulator pipeline does not lead to any extra delays in our write-back design because, due to the dependencies described above, the pipeline must be flushed when force calculation on a new i atom is to begin anyway. Note that calculating the forces on the j atoms does not require an accumulator, only an adder, because the force on a j atom is updated at most once for any given i atom.

We use Carte version 2.1 together with the Xilinx ISE tools version 7.2.3 and the Intel C compiler version 8.1 to generate the executable. We use only one of the two Xilinx Virtex-II XC2V6000-4 FPGAs on the MAP. The FPGA area of the nonbonded force calculation is 33,634 slices, which is 99% of the available slices. 28% of the available block RAMs and 77% of the available multipliers are used. The required 100 MHz frequency is met.

4.1 Performance

We ran each of the simulations on the MAPstation for 1000 steps and compared that to the same simulations running in software only on the MAPstation. Table 2 shows the results. For the palmitic acid simulation, we achieve a $2\times$ speed-up while for the CheY simulation, we achieve a $1.9\times$ speed-up. This is noteworthy because we achieve these speed-ups despite leaving parts of the simulation in software and not using the GPP node and RH node in parallel. The difference in speed-up between the two simulations is due to the tasks left in software accounting for a greater percentage of the simulation time than the task mapped to hardware in the CheY simulation than in the palmitic acid simulation (see Table 1).

The $2\times$ speed-up for palmitic acid is the same as was predicted in Section 3.3. The time for the simulation was slightly longer than predicted, likely due to overheads unaccounted for in the performance modeling, such as DMA start-up costs and the cost of transferring the first sections of the neighbor list.

These speed-ups are influenced by the parameters of the simulation. For example, were we to build the neighbor

Table 2. MAPstation performance results

Simulation	Latency (s/step)		Speed-up
	SW only	SW + HW	
Palmitic Acid	1.28	0.66	$2.0\times$
CheY Protein	0.74	0.39	$1.9\times$

list less frequently than once every 10 steps, the speed-ups achieved would be greater because building the neighbor list would account for less of the overall simulation time. The neighbor list cutoff distance can also affect the application’s profile, as can other physical properties.

4.2 Scalability

In the purely software domain, the scalability of MD simulations is judged by the speed-ups obtained by spreading the simulation over multiple processors, where each processor executes all tasks in the velocity Verlet algorithm. This type of scaling is certainly possible with reconfigurable computers. Indeed, we comment on it in Section 6. Here, we focus on a different kind of scalability: having one GPP node but multiple RH nodes.

As in the current implementation, all tasks in the simulation except for nonbonded force calculation are executed on the GPP node. The constants are communicated to all of the RH nodes during the first step of the simulation. At each step of the simulation, the position and charge data are transferred to all of the RH nodes. If there are R RH nodes, then each one is given approximately $1/R$ of the neighbor list to process. Upon completion of calculating the nonbonded forces for its portion of the neighbor list, each RH node then transmits the forces back to the GPP node. For each atom i , the GPP node calculates $\vec{f}_i = \sum_{r=1}^R \vec{f}_{i_r}$, where \vec{f}_{i_r} is the force on atom i calculated by RH node r .

Figure 5 shows the estimated time per step as the number of RH nodes increases from 0 to 4. Equation 5 was used to produce the estimates. Note that the communication time is so small that it is barely visible. We ignore the cost of the final summation mentioned above and of deciding how to split the neighbor list among the RH nodes, as the time for these computations will be dwarfed by the other software tasks. Clearly, once more than 2 RH nodes are employed, the tasks in software become dominant.

$$T = T_{\text{SWtasks}} + RT_{\text{comm}} + \frac{T_{\text{RH}}}{R}. \tag{5}$$

5 Comparison to Related Work

Several prior works have studied implementing tasks in MD simulations in hardware [13, 14, 17]. However, these

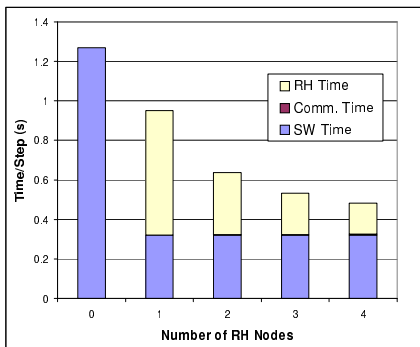


Figure 5. Estimates for the performance of the design when using 0 to 4 RH nodes

works do not consider the MD simulation application as a whole. So, we instead focus our comparisons on the two works in which most of the major tasks in an MD simulation have been implemented.

One such work is [3]. In this work, the velocity Verlet algorithm is implemented on a hardware platform with four FPGAs, memory, and interconnect. All steps of the velocity Verlet algorithm are implemented in hardware. Only the Lennard-Jones force is considered. All the computations are done in fixed-point arithmetic. The Lennard-Jones force is calculated with table look-up and interpolation. The system uses the $O(n^2)$ method to find interacting pairs.

[7] is another work in which the entire velocity Verlet algorithm is considered. This work is similar to [3] in that both implementations perform all steps of the algorithm in hardware using fixed-point arithmetic, only nonbonded force calculation is implemented, this nonbonded force calculation is performed with table lookup and interpolation, and, it seems, the interaction pairs are found using the $O(n^2)$ approach. One advance in [7] is the addition of cutoff Coulomb force calculation. Another advance is support for multiple types of atoms. Two types can be accommodated in hardware; more than two types requires swapping in tables from external memory. Yet another advance is the use of multiple pipelines in parallel in the design.

Our hardware/software approach to accelerating MD simulations is fundamentally different than those of [3] and [7]. These implementations move all steps of the simulation—position update, velocity update, acceleration update, and force calculation—to hardware. We, on the other hand, have left most of the simulation in software, moving only the nonbonded force calculation to hardware. Further, rather than switching to a fixed-point implementation, we maintain floating-point arithmetic throughout, in both hardware and software. Another difference is in the way we approach the calculation of the nonbonded forces. Both of the other implementations use table lookup and interpolation while ours directly calculates the forces from their equations.

Our approach gives us the advantage over the previous implementations that it is simple to integrate tasks that are well-suited for software. For example, we have implemented (in software) the bonded force calculation, which the earlier implementations have not. Further, integrating other advanced tasks is easily accomplished in our approach: it is merely a matter of integrating new function calls in the software.

Our approach has also made it easy to handle multiple types of atoms. Since we use direct calculation of the forces, we need not use any block RAMs for table-lookup in the force calculation. Further, the number of type-based constants that need to be stored is only four times the square of the number of types, as opposed to needing separate force calculation tables for interactions for various types of atoms, as is the case in [7]. Thus we can handle a large number of types of atoms. The use of only a small number of block RAMs for constants is crucial in our approach because it keeps the rest of the block RAMs available for the intermediate storage needs of our write-back design.

Finally, our approach has facilitated the use of the neighbor list technique for finding interacting pairs of atoms. This is very important because it drastically reduces the number of pairs that need to be evaluated at each time step. In the $O(n^2)$ approach, most of the time is spent finding distances between pairs of atoms that are too far apart to necessitate the calculation of the forces they exert on one another. Even with the advantages of hardware, it is difficult for a system using this approach to scale to large simulations; software using more advanced techniques such as the neighbor list or the linked-cell list will likely be faster.

Table 3 shows the speed-ups over software baselines for the design proposed here and the fastest speed-ups reported in [3] and [7]. Note that the speed-ups reported in [3] and [7] are much greater than that reported for the proposed design, but that those speed-ups are obtained by comparing against software baselines that are much slower than our software implementation, despite having fewer atoms and not calculating bonded forces. Also note that bonded force calculation and potential energy calculation are not done in [3] and [7]. [3] and [7] do, however, use varying-precision fixed-point arithmetic to provide results nearly as accurate as double-precision floating-point arithmetic, while the accuracy of our simulation is limited to single-precision floating-point arithmetic.

6 Conclusion

In this paper, we have presented an implementation of molecular dynamics simulations on a reconfigurable computer that achieves a $2\times$ speed-up over the corresponding software-only solution. The approach taken here, which is different than that of prior work, is to tightly integrate

Table 3. Performance Comparison

	Proposed	[3]	[7]
Number of atoms	52558	8192	8192
SW baseline latency (s/step)	1.3	10.8	9.5
Speed-up	2×	21×	89×

the general purpose processor and reconfigurable hardware, having the reconfigurable hardware perform the nonbonded force calculation and the general purpose processor perform the rest of the tasks in the simulation. This is different than hybrid design approaches, as in [18], where the general purpose processor and reconfigurable hardware work together on a common task. Here, the division is at the task level. Using this approach allows us to handle large, real-world simulations and gives the flexibility to easily add features to our implementation. Our work shows that significant speed-ups can be obtained with this approach, even though only a portion of the problem is accelerated. The main limitations to the proposed MD system are the use of the cutoff Coulomb method and the use of single-precision arithmetic. The latter is due to limits in the size of the FPGA and memory available in our target machine rather than any inherent feature of our design.

There are several future directions to pursue. One is to investigate possibilities for implementing double-precision simulations. While this does not seem possible with the currently available reconfigurable computer considered in this paper, it may be possible on future reconfigurable computers that have more on-board memory banks and larger FPGAs. Another direction is to study parallel implementations containing multiple GPP nodes and RH nodes, e.g., a MAPstation with multiple general purpose processors, each connected to a MAP processor. When MD simulations are parallelized over multiple processors, the linked-cell list is sometimes used over the neighbor list because it is easier to account for atoms moving between processors. We will need to determine the best setup for such a simulation.

Acknowledgments

We wish to thank the U. S. Army ERDC MSRC for access to its MAPstation and Steve Heistand of SRC Computers for his assistance with the implementation.

References

[1] M. Allen and D. J. Tildeseley. *Computer Simulation of Liquids*. Oxford University Press, New York, 1987.

[2] *The Amber Molecular Dynamics Package*. <http://amber.scripps.edu/>.

[3] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow. Reconfigurable molecular dynamics simulator. In *Proceed-*

ings of the IEEE Symposium on Field-Programmable Custom Computing Machines, pages 197–206, April 2004.

[4] P. Belanovic and M. Leeser. A library of parameterized floating point modules and their use. In *Proceedings of the 12th International Conference on Field Programmable Logic and Applications*, September 2002.

[5] J. Detrey and F. de Dinechin. A VHDL library of parametrizable floating-point and LNS operators for FPGA. <http://perso.ens-lyon.fr/jeremie.detrey/FPLibrary/>.

[6] G. Govindu, R. Scrofano, and V. K. Prasanna. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2005.

[7] Y. Gu, T. VanCourt, and M. C. Herbordt. Accelerating molecular dynamics simulations with configurable circuits. In *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications*, August 2005.

[8] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *Journal of Computational Physics*, 151:283–312, 1999.

[9] OProfile. <http://oprofile.sourceforge.net/>.

[10] M. Patra, M. Karttunen, M. T. Hyvnen, E. Falck, P. Lindqvist, and I. Vattulainen. Molecular dynamics simulations of lipid bilayers: Major artifacts due to truncating electrostatic interactions. *Biophysics Journal*, 84:3636–3645, 2003.

[11] *PDB Format Guide*. http://www.rcsb.org/pdb/file_formats/pdb/pdbguide2.2/guide2.2_frame.htm%1.

[12] L. Phillips, R. Sinkovits, E. Oran, and J. Boris. The interaction of shocks and defects in Lennard-Jones crystals. *Journal of Physics: Condensed Matter*, 5(35):6357–6376, August 1993.

[13] R. Scrofano and V. K. Prasanna. Computing Lennard-Jones potentials and forces with reconfigurable hardware. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2004.

[14] M. C. Smith, J. S. Vetter, and X. Liang. Accelerating scientific applications with the SRC-6 reconfigurable computer: Methodologies and analysis. In *Proceedings of the 2005 Reconfigurable Architectures Workshop*, April 2005.

[15] SRC Computers, Inc. <http://www.srccomputers.com>.

[16] D. van der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. C. Berendsen. GROMACS: Fast, flexible, and free. *Journal of Computational Chemistry*, 26(16):1701–1718, December 2005.

[17] C. Wolinski, F. Trouw, and M. Gokhale. A preliminary study of molecular dynamics on reconfigurable computers. In *Proceedings of the 2003 International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2003.

[18] L. Zhuo and V. K. Prasanna. Scalable hybrid designs for linear algebra on reconfigurable computing systems. In *Proceedings of the 12th International Conference on Parallel and Distributed Systems*, 2006. (submitted).