FRIEDRICH ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
Lehrstuhl Informatik 10 (Systemsimulation) • Prof. Dr. Ulrich Rüde
Cauerstraße 6 • D-91058 Erlangen

Bachelor Thesis

Optimization and Performance Analysis of the Lattice Boltzmann Method on x86-64 based Architectures

Simon Hausmann

Erlangen, 28. April 2005

Optimization and Performance Analysis of the Lattice Boltzmann Method on x86-64 based Architectures

Simon Hausmann

Bachelor Thesis

Aufgabensteller:	Prof. Dr. Ulrich Rüde
Betreuer:	Dipl.-Ing. Jan Treibig
Bearbeitungszeitraum:	November 2004 - April 2005

Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 28. April 2005 .....

Abstract

The Lattice Boltzmann method (LBM) is a well established technique to simulate fluids. With a model based on cellular automata it divides the computational domain into a regular grid of cells. Fast processing of cells on a grid in turn requires fast access to main memory, which is usually not given compared to the processing power of on modern computers with high clock frequencies. In addition the algorithm for the fluid flow simulation is arithmetically very demanding. Fortunately the LBM also offers many opportunities to apply common memory optimization techniques.

This thesis will show how effectively established general cache optimizations combined with architecture specific techniques can reduce the overhead of slow main memory access. Theoretical investigations and comparison of results show that arithmetic coding of the algorithm becomes a visible performance limiting factor.

Additionally the thesis covers benchmarking techniques to determine the limits and properties of the cache and memory subsystem.

Contents

1 Introduction

Computer architectures continue to improve at a fast pace. At the same time the demand for computer based simulation of real-life problems grows, for example in car crash simulation or wheather forecasting. Bigger domains need to be calculated, more complex situations modelled and approximated.

As an example the simulation of fluid dynamics results in very compute intensive efforts, and as such they are a popular subject of research for improved performance. The underlying physical model of computational fluid dynamics (CFD) are the *Navier-Stokes* equations. The standard approach for simulation is their discretization using methods of finite volumes, finite elements or finite differences combined with numerical solving of the resulting system of linear equations.

The *Lattice Boltzmann method* is an alternative approach to CFD that combines the essential properties of the Navier-Stokes equations, such as mass, energy and momentum conservation, with a different underlying particle-based model. A big advantage is that the core algorithm is straightforward to implement from the formulas. Also features like obstacles in the fluid are easier to implement than with the Navier-Stokes discretization approach. On the other hand the method's indirect description of the fluid flow by distribution functions creates a certain distance to the concrete physical equations. It can be difficult to combine with other physical simulations.

This thesis starts with a brief introduction to the algorithm. Chapter 3 focuses on the implementation on the x86-64 architecture, with regards to the limits the instruction set, the microarchitecture and the algorithm imposes. In chapter 4 the analysis is extended to cache optimization, with techniques such as 4-way cache blocking, software prefetching and large memory pages. Measurements on implementations of the x86-64 architecture are shown in chapter 5 and put into relation to each other. Final conclusions will be presented in chapter 6, as well as suggestions for further research work.

2 The Lattice Boltzmann Method

This chapter presents the core Lattice Boltzmann algorithm. The individual steps are given in their mathematical form in the first section as well as in example C code for implementation in the second section. In addition the test problem for verification of the simulation is discussed, along with the modifications to the implementation it requires.

2.1 The algorithm in 3D

In the following a brief summary of the Lattice Boltzmann method is given. The formulas result from specializing the general equations to three dimensions. A detailed description along with a deep theoretical background can be found in [WG00].

The Lattice Boltzmann method numerically simulates the flow of fluid, based on the model of cellular automaton. The computational domain is mapped onto a regular grid with cells, the lattice. Each cell represents a volume element of fluid particles. The motion of the particles in the fluid is described only indirectly by distribution functions. A cell is divided into discrete directions of velocity, and for each a particle distribution function is defined, representing the motion of particles into that direction. These functions form the state of the cell.

As discrete time advances the flow of particles is simulated by moving the values of the distribution functions to the neighboring cells and calculating the collision with particles from other directions.

The model for the discrete directions of velocity used in this thesis is the so-called *D3Q19* model (see Fig. 2.1) that defines 19 directions. The abbreviated directions in the figure are defined as C (center), N (north), S (south), W (west), E (east), T (top), B (bottom), NW (northwest), NE (northeast), SW (southwest), SE (southeast), TN (top north), TS (top south), TW (top west), TE (top east), BN (bottom north), BS (bottom south), BW (bottom west), and BE (bottom east).

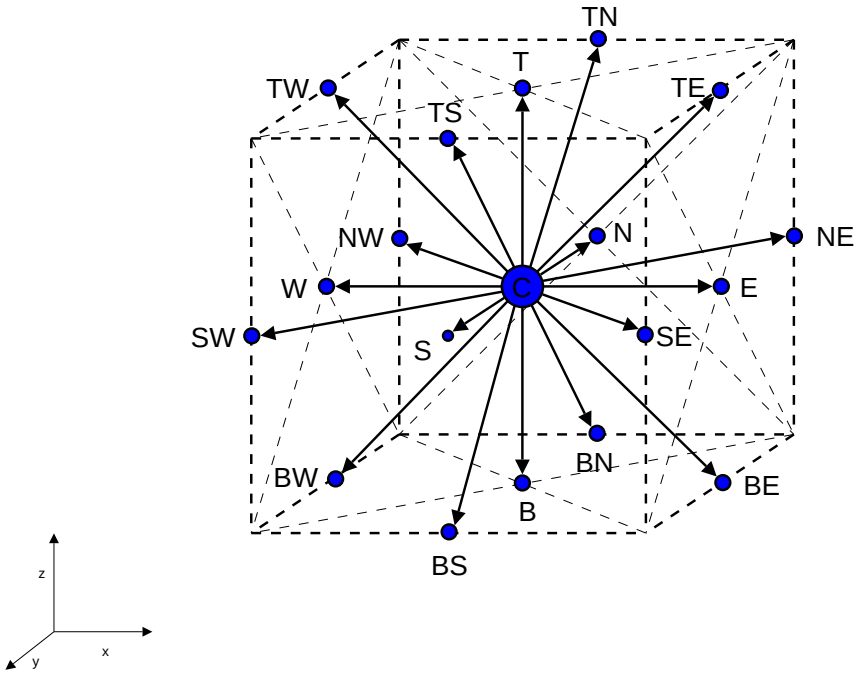


Figure 2.1: Lattice cell with 19 discrete directions in the D3Q19 model.

These 19 discrete directions of velocity $\vec{c}_i$ are defined as follows:

$$\vec{c}_i := \begin{cases} (0,0,0) & i = \text{C}, \\ (\pm 1,0,0) & i = \text{W, E}, \\ (0,\pm 1,0) & i = \text{N, S}, \\ (0,0,\pm 1) & i = \text{T, B}, \\ (\pm 1,\pm 1,0) & i = \text{NW, NE, SW, SE}, \\ (\pm 1,0,\pm 1) & i = \text{TW, TE, BW, BE}, \\ (0,\pm 1,\pm 1) & i = \text{TN, TS, BN, BS}. \end{cases}$$

With $\vec{x}$ denoting the position in the grid and t the time step parameter, the initial cell state is defined using the following values for the particle distribution functions:

$$F_i(\vec{x}, t = 0) := \begin{cases} 1/3 & i = C, \\ 1/18 & i = W, E, N, S, T, B, \\ 1/36 & i = NW, NE, SW, SE, TW, TE, BW, BE, TN, TS, BN, BS. \end{cases}$$

From the distribution functions F_i and the discrete cell velocities the *mass density* ρ and *fluid velocity* $\vec{u}$ can be calculated:

$$\rho(\vec{x},t) := \sum_i F_i(\vec{x},t) \tag{2.1}$$

$$\vec{u}(\vec{x},t) := \frac{1}{\rho(\vec{x},t)} \sum_i F_i(\vec{x},t)\ \vec{c}_i \tag{2.2}$$

From the equidistant cell size Δx and the time step Δt the lattice speed s is defined as

$$s = \frac{\Delta x}{\Delta t}.$$

For each direction in a cell a local equilibrium distribution function is defined. For $i = \mathrm{C}$:

$$F_i^{(eq)}(\vec{x},t) = \frac{\rho(\vec{x},t)}{3} \cdot \left[1 - \frac{3}{2}\frac{\langle \vec{u}(\vec{x},t), \vec{u}(\vec{x},t)\rangle}{s^2}\right] \tag{2.3}$$

For $i \in \{\mathrm{W},\mathrm{E},\mathrm{N},\mathrm{S},\mathrm{T},\mathrm{B}\}$:

$$F_i^{(eq)}(\vec{x},t) = \frac{\rho(\vec{x},t)}{18} \cdot \left[1 + 3\frac{\langle \vec{c}_i, \vec{u}(\vec{x},t)\rangle}{s^2} + \frac{9}{2}\frac{\langle \vec{c}_i, \vec{u}(\vec{x},t)\rangle^2}{s^4} - \frac{3}{2}\frac{\langle \vec{u}(\vec{x},t), \vec{u}(\vec{x},t)\rangle}{s^2}\right] \tag{2.4}$$

For $i \in \{\mathrm{NW},\mathrm{NE},\mathrm{SW},\mathrm{SE},\mathrm{TW},\mathrm{TE},\mathrm{BW},\mathrm{BE},\mathrm{TN},\mathrm{TS},\mathrm{BN},\mathrm{BS}.\}$:

$$F_i^{(eq)}(\vec{x},t) = \frac{\rho(\vec{x},t)}{36} \cdot \left[1 + 3\frac{\langle \vec{c}_i, \vec{u}(\vec{x},t)\rangle}{s^2} + \frac{9}{2}\frac{\langle \vec{c}_i, \vec{u}(\vec{x},t)\rangle^2}{s^4} - \frac{3}{2}\frac{\langle \vec{u}(\vec{x},t), \vec{u}(\vec{x},t)\rangle}{s^2}\right] \tag{2.5}$$

These functions are used in the *Lattice Boltzmann Equations*:

$$\tilde{F}_i(\vec{x},t+\Delta t) = F_i(\vec{x} - \vec{c}_i\Delta t, t) \tag{2.6}$$

$$F_i(\vec{x},t+\Delta t) = (1-\omega)\tilde{F}_i(\vec{x},t+\Delta t) + \omega\tilde{F}_i^{(eq)}(\vec{x},t+\Delta t) \tag{2.7}$$

Equation (2.6) is called the stream step, which defines the cell neighborhood that leads to the new state, assuming a flow of particles. Equation (2.7) then defines that the new state for each particle distribution function is the sum of the streamed non-equilibrium distribution function $\tilde{F}_i(\vec{x},t)$ and the streamed local distribution function $\tilde{F}_i^{(0)}$, weighted by the factor ω. This is called the collide step, as it simulates the collision of the particles as a result of fluid flow.

Whether the stream step (equation (2.6)) is evaluated before the collide step (equation (2.7)) or the other way around is not relevant to the ultimate result of the simulation. The speed of execution may differ though, but that is only due to architecture specific implementation details.

2.2 Implementation

2.2.1 Basic Definitions

The grid of the simulation consists of a three-dimensional array of cells. Each cell consists of 19 values for the particle distribution function, stored as double precision floating point numbers. The cells and therefore also the cell values lie consecutively in memory, in the same order as the x, y and z indices run over the grid in the main iteration loop. This is the same layout as in [Igl03].

For the update of one cell the values of the particle distribution function of all surrounding cells from the *previous* time step are needed. Therefore it is not possible to operate entirely in one grid, as otherwise the neighborhood of a cell would consists partly of cells from the previous timestep as well as cells from the current time step. This would yield wrong results. Therefore two separate grid arrays are needed. [Wil03] shows an alternative solution by merging the two grids. In the example listings for the sake of simplicity two separate grids are used.

Listing 2.1 shows a skeleton of C code that contains the basic data type definitions as well as the main iteration loop with comments denoting the order of calculation.

Listing 2.1: Basic skeleton of the algorithm

```
enum Directions {
    Center = 0, North, South, West, East, Top, Bottom,
    NorthWest, NorthEast, SouthWest, SouthEast, TopNorth,
    TopSouth, TopWest, TopEast, BottomNorth, BottomSouth,
    BottomWest, BottomEast,
    NumDirections = 19
};
double grid1[GridDepth][GridHeight][GridWidth][NumDirections];
double grid2[GridDepth][GridHeight][GridWidth][NumDirections];
int x, y, z;

for (z = 0; z < GridDepth; ++z)
    for (y = 0; y < GridHeight; ++y)
        for (x = 0; x < GridWidth; ++x) {
            // gather particle distribution functions values for
            // all directions from surrounding cells of grid1
            ...
            // calculate mass density rho
            ...
            // calculate components ux, uy and uz of velocity
            ...
            // calculate local equilibrium distribution function
            // using rho and u components
            ...
            // calculate values of the particle distribution function
            // at new time step as weighted sum of the pdf from the
            // old step and the local equilibrium pdf. store them in
            // grid2
            ...
        }
```

2.2.2 The Collide Step

The collide step can be implemented by coding the equations from (2.1) straight down to (2.7). Listings 2.2 and 2.3 show the implementation. They assume that the values of the particle distribution function for the *current* cell are located in variables (or macros) called `Cell_`*Direction*. For example `Cell_Center` corresponds to $F_{\mathrm{C}}(\vec{x}, t)$, or `Cell_North` to $F_{\mathrm{N}}(\vec{x}, t)$.

Listing 2.2: C implementation of the first part of collide step that calculates ρ and $\vec{u}$

```
double rho, ux, uy, uz;

rho = Cell_Center + Cell_North + Cell_South + Cell_West
      + Cell_East + Cell_Top + Cell_Bottom + Cell_NorthWest
      + Cell_NorthEast + Cell_SouthWest + Cell_SouthEast + Cell_TopNorth
      + Cell_TopSouth + Cell_TopWest + Cell_TopEast + Cell_BottomNorth
      + Cell_BottomSouth + Cell_BottomWest + Cell_BottomEast;

ux = Cell_East + Cell_NorthEast + Cell_SouthEast
     + Cell_TopEast + Cell_BottomEast
     - Cell_West - Cell_NorthWest - Cell_SouthWest
     - Cell_TopWest - Cell_BottomWest;

uy = Cell_North + Cell_NorthEast + Cell_NorthWest
     + Cell_TopNorth + Cell_BottomNorth
     - Cell_South - Cell_SouthEast - Cell_SouthWest
     - Cell_TopSouth - Cell_BottomSouth;

uz = Cell_Top + Cell_TopNorth + Cell_TopSouth
     + Cell_TopWest + Cell_TopEast
     - Cell_Bottom - Cell_BottomNorth - Cell_BottomSouth
     - Cell_BottomWest - Cell_BottomEast;

ux /= rho;
uy /= rho;
uz /= rho;
```

Listing 2.3: Excerpts of a C implementation of the second part of collide step that calculates the local distribution functions and resulting new values for the particle distribution functions, unoptimized.

```
 // values for the local equilibrium distribution
double F[NumDirections];

F[Center] = (1.0 / 3.0) * rho * (1.0
                                 - 1.5 * (sqr(ux) + sqr(uy) + sqr(uz)));

F[North] = (1.0 / 18.0) * rho * (1.0 + (3.0 * uy)
                                 + ((9.0 / 2.0) * uy * uy)
                                 - 1.5 * (sqr(ux) + sqr(uy) + sqr(uz)));

F[South] = (1.0 / 18.0) * rho * (1.0
                                 - (3.0 * uy)
                                 + ((9.0 / 2.0) * uy * uy)
                                 - 1.5 * (sqr(ux) + sqr(uy) + sqr(uz)));
```

```
15
16 F[West] = (1.0 / 18.0) * rho * (1.0
17                                 - (3.0 * ux)
18                                 + ((9.0 / 2.0) * ux * ux)
19                                 - 1.5 * (sqr(ux) + sqr(uy) + sqr(uz)));
20 ...
21 // calculate values for particle distribution functions in new time step
22 New_Cell_Center = (1.0 - omega) * Cell_Center + omega * F[Center];
23 New_Cell_North  = (1.0 - omega) * Cell_North + omega * F[North];
24 New_Cell_South  = (1.0 - omega) * Cell_South + omega * F[South];
25 New_Cell_West   = (1.0 - omega) * Cell_West + omega * F[West];
26 ...
```

2.2.3 The Streaming Step

The streaming step (equation (2.6)) represents the flow of particles across cell boundaries. The inputs to the calculation of the *mass density* and *velocity* in the collide step are the particle distribution function values from the surrounding cells. For each direction the value of the function is taken from the neighboring cell in the *opposite* direction. With regards to the listings from section 2.2.2 this means that the `Cell_`*Direction* macros point to corresponding entries in the surrounding cells and the `New_Cell_`*Direction* macros point to the current cell, as per listing 2.4. This is the *stream-collide* order, as values are *gathered* first. If *collide-stream* is used instead the macros are reversed, with `Cell_`*Direction* pointing to the current cell and `New_Cell_`*Direction* pointing to the surrounding ones, so that the new values are *pushed* to the neighborhood (see listing 2.5). Figure 2.2 illustrates the flow of particles from the surrounding cells.

Listing 2.4: Example implementation of the Cell macros for stream-collide order

```
1  // the discrete velocity direction vectors
2  #define Direction_Center_X 0
3  #define Direction_Center_Y 0
4  #define Direction_Center_Z 0
5
6  #define Direction_North_X 0
7  #define Direction_North_Y -1
8  #define Direction_North_Z 0
9
10 #define Direction_South_X 0
11 #define Direction_South_Y +1
12 #define Direction_South_Z 0
13 ...
14 #define Cell_Center grid1[z][y][x][Center]
15
16 #define Cell_North grid1[z - Direction_North_Z] \
17                        [y - Direction_North_Y] \
18                        [x - Direction_North_X] \
19                        [North]
20
21 #define Cell_South grid1[z - Direction_South_Z] \
22                        [y - Direction_South_Y] \
```

```
                        [x - Direction_South_Y] \
                        [South]

#define New_Cell_Center grid2[z][y][x][Center]
#define New_Cell_North grid2[z][y][x][North]
#define New_Cell_South grid2[z][y][x][South]
```

Listing 2.5: Example implementation of the Cell macros for collide-stream order

```
#define Cell_Center grid1[z][y][x][Center]
#define Cell_North grid1[z][y][x][North]
#define Cell_South grid1[z][y][x][South]
...
#define New_Cell_Center grid2[z][y][x][Center]

#define New_Cell_North grid2[z - Direction_North_Z] \
                            [y - Direction_North_Y] \
                            [x - Direction_North_X] \
                            [North]

#define New_Cell_South grid2[z - Direction_South_Z] \
                            [y - Direction_South_Y] \
                            [x - Direction_South_Y] \
                            [South]
```

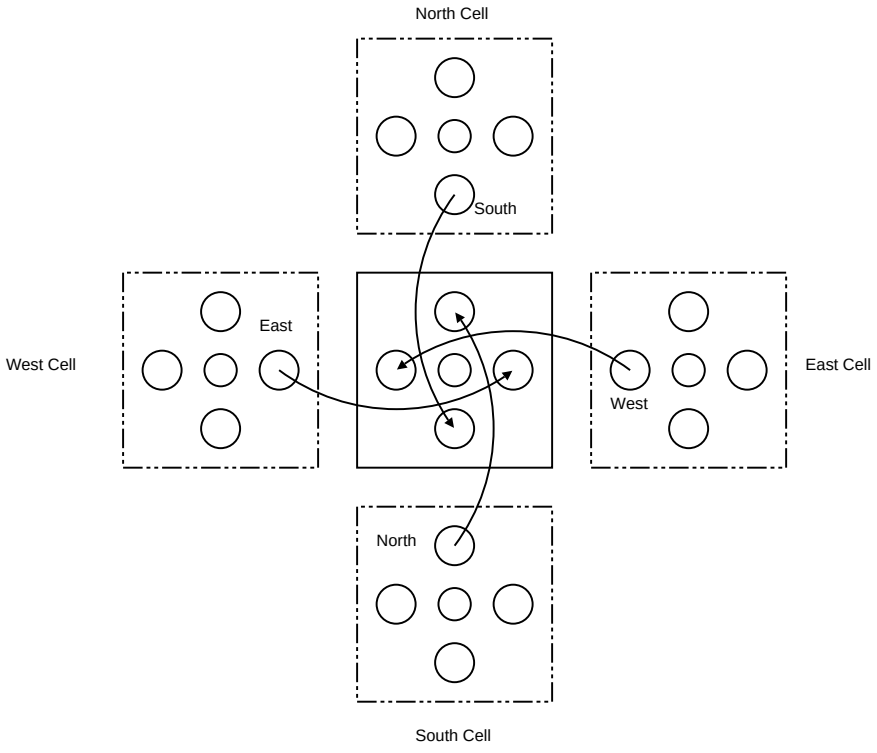


Figure 2.2: Simplified view of the process of gathering the particle distribution function values from surrounding cells, as part of the streaming step.

2.3 Lid Driven Cavity as Test Problem

As in [Igl03] and [Don04] the lid driven cavity problem served as test case for the performance measurements as well as for the verification of the solver.

It consists of a cube filled with fluid. One side of the cube serves as acceleration plane by sliding constantly. This puts the fluid into rotation, as one can see on the velocities of the fluid in the cube in Fig. 2.3. The acceleration is implemented by assigning the cells in the acceleration area a constant velocity $\vec{u}$. This change also requires extending the basic solver by introducing another three dimensional array that allows distinguishing between three types of cells: Regular fluid, a special acceleration cell or a boundary.

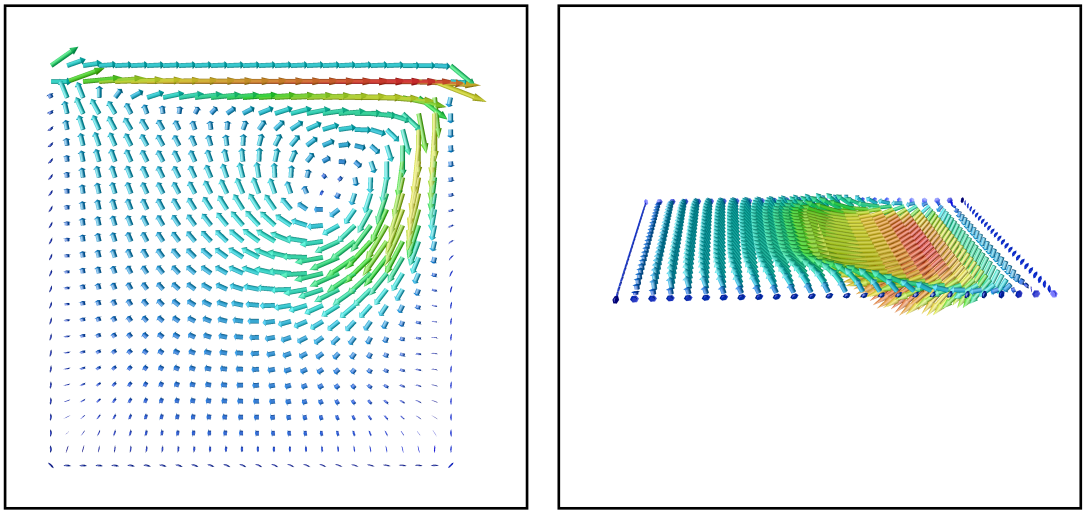


Figure 2.3: Two cuts through three dimensional lid driven cavity, showing the $\vec{u}$ velocity vectors. Visualization using OpenDX.

For cells of type *Boundary* the so-called *no-slip* behavior is implemented. By reversing the velocities of the particles that hit the boundary the average velocity *at* the boundary becomes zero (see listing 2.6).

Listing 2.6: Excerpts of the implementation of no-slip boundary conditions

```
...
if (cellTypes[z][y][x] == Obstacle) {
    New_Cell_North = Cell_South;
    New_Cell_South = Cell_North;
    New_Cell_West = Cell_East;
    New_Cell_East = Cell_West;
    New_Cell_Top = Cell_Bottom;
    New_Cell_Bottom = Cell_Top;
    ...
}
...
```

The introduction of the boundary cells required another change in the solver: As the outermost cells are marked as boundary and the *no-slip* handling still includes the streaming it may happen that cells outside the grid are accessed. In order to avoid extra conditional handling of these cases another extra layer of cells is put around the whole grid, the so-called *ghost layer*. This means that the grid is enlarged by two more cells in each dimension. However these extra cells are not actually processed as part of the main iteration, they merely exist so that read or write access into that area does not result in out-of-bounds array access. Therefore in addition to the grid extension the iteration bounds need to be adjusted, as shown in listing 2.7.

Listing 2.7: New loop bounds for ghost-layer handling

```
1 double grid1[GridDepth+2][GridHeight+2][GridWidth+2][NumDirections];
2 double grid2[GridDepth+2][GridHeight+2][GridWidth+2][NumDirections];
3 for (z = 1; z < GridDepth + 1; ++z)
4     for (y = 1; y < GridDepth + 1; ++y)
5         for (x = 1; x < GridDepth + 1; ++x)
6             ... // Lattice Boltzmann step here
```

3 Arithmetic Analysis

This chapter takes a look at how the Lattice Boltzmann algorithm can be implemented on a x86-64 based processor, after an introduction of the underlying architecture. It is examined how the algorithm can be mapped to the pseudo-vector extensions of the instruction set and how calculation dependencies in the algorithm can affect performance. In the last section the relation of the algorithm to a processor's maximal performance is discussed.

3.1 Short Overview of the x86-64 Architecture

The x86-64 architecture is a superset of the widely deployed x86 architecture. It is available in newer revisions of Intel's Pentium 4 and Xeon processors as well as in AMD's Athlon-64 and Opteron processors. The following is just a brief summary of its features that are relevant for this thesis. More detailed information can be found in [Dev03] and [Cor04b].

The previous x86 architecture has only 8 general purpose registers, two of which are needed for stack management. So usually only 6 are available for tasks like integer arithmetic, array indexing or loop counting. For the Lattice Boltzmann algorithm in 3D with at least three nested loops and complex array indexing this means that the compiler has to emit a lot of additional code besides the computational kernel. For example code for storing temporary variables on the stack in main memory to free up registers (*register spilling*). A key enhancement of x86-64 is that there are 16 general purpose registers (GPRs) available, each 64 bits wide (see Fig. 3.2). This results in shorter, simpler and therefore faster code.

Intel's SSE2[1] feature is also part of the x86-64 instruction set. It is the recommended way of coding floating point calculations, featuring another 16 registers (`xmm0-15`). They can hold integer or floating point values, with single or double precision. Given the width of 128 bits one register can store multiple values of the same type. Such *packed values* can be used with *packed operations* that result in one instruction operating multiple times on independent data (see Fig. 3.1). Current processors do not implement such operations in parallel as they lack multiple multiplication or addition units. But this is possible to change in the future. In any case *packed operations* permit the instruction scheduler to overall see more macro operations and get a better overview of dependencies between instructions.

As the general purpose registers are 64 bits wide the address space for main memory

[1] Streaming SIMD Extension

is extended to the same width, theoretically 2^{64} bytes. Current implementations of the x86-64 architecture however use only 48-bits for the virtual and 40 bits for the physical addressing. With regards to computational fluid dynamics this means that much bigger domains can be simulated on one machine. For example a 250^3 grid with 19 doubles per cell takes up $250^3 \cdot 19 \cdot 8$ Byte, approximately 2.2 GByte. With the use of two grids the logical address space of 4 Gigabytes on a x86 machine is already exceeded then, a larger address space as the one the x86-64 architecture provides is required.

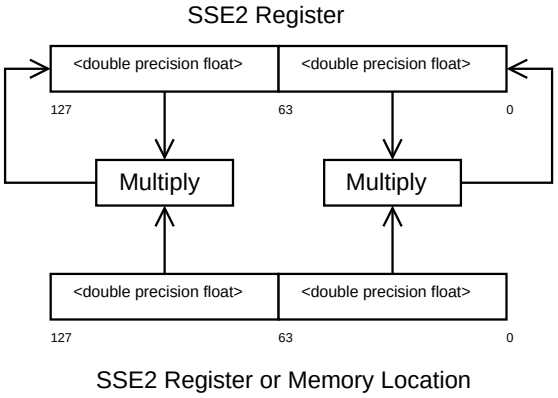


Figure 3.1: SSE2 Packed Multiplication: The two separate multiplications on *doubles* are coded using one single instruction.

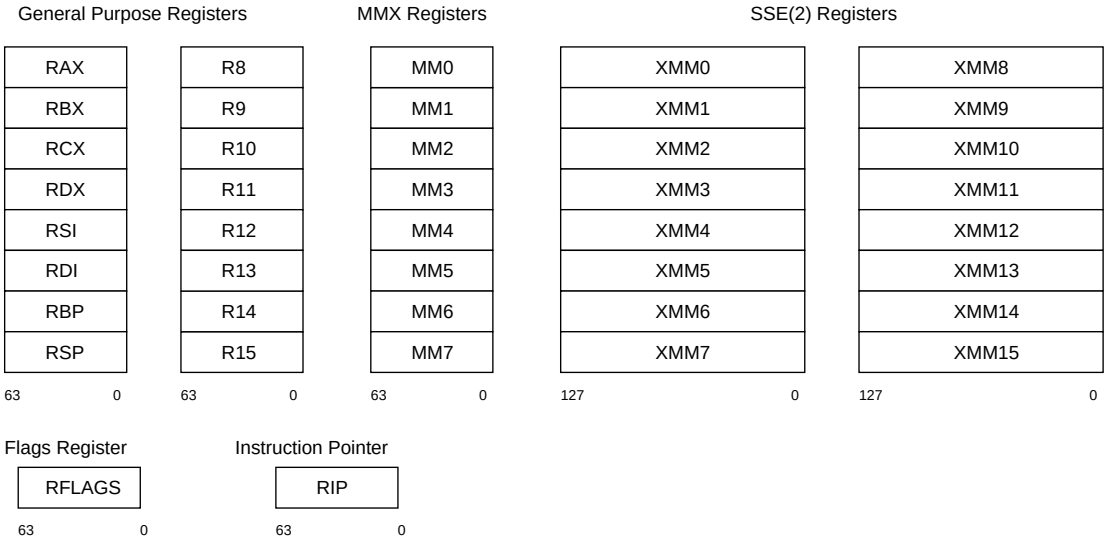


Figure 3.2: Schematic view of x86-64 register set.

3.2 Calculation Dependencies in the Collide Step

The arithmetic kernel of the algorithm is the collide step. The collision equation (2.7) is evaluated multiple times for each direction of the current cell. The evaluation depends on the previous calculation of the mass density and the velocity as well as the local equilibrium distribution function ((2.3), (2.4), (2.5)). Figure 3.3 illustrates the flow of values. One can see that the new particle distribution functions do not depend on each other. Their calculation sequences have two important properties in common though:

1. The formulas of the local equilibrium distribution functions for $i \in \{W, E, N, S, T, B\}$ and $i \in \{NW, NE, SW, SE, TW, TE, BW, BE, TN, TS, BN, BS.\}$ differ only in their leading constant factors and signs inside, caused by velocity direction vectors $\vec{c}_i$. Otherwise the operations are the same between the directions.
2. The weighting of old particle distribution function and local equilibrium distribution function in Eqn. (2.7) maps to the same sequence of operations for all directions.

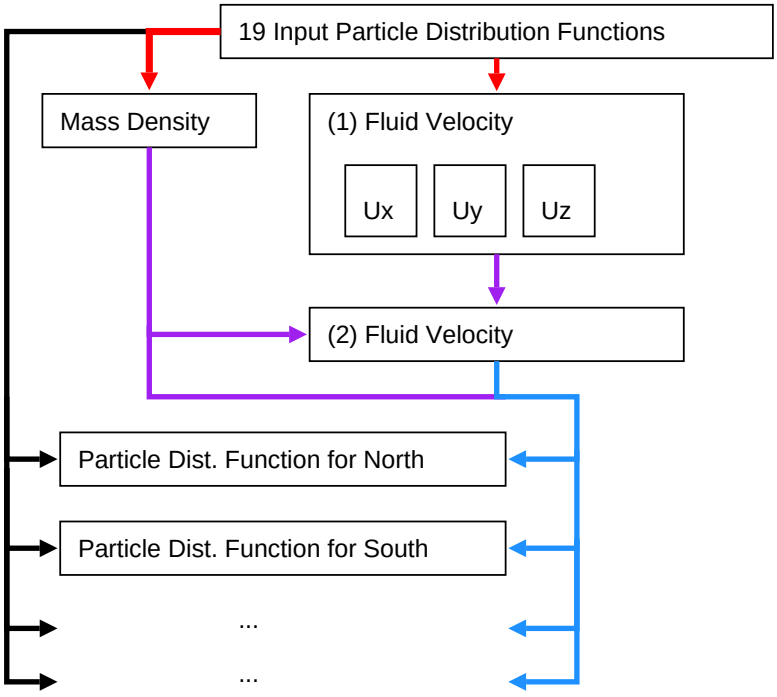


Figure 3.3: Graph showing the flow of calculated values in the collide step.

With regards to the x86-64 instruction set architecture these two properties permit combining always two particle distribution function updates into one sequence of instructions by the use of SSE2 *packed operations*. This reduces the amount of instructions needed to formulate the algorithm and results in faster execution.

An important property of the Athlon-64 and Opteron processors is that they have one unit for floating point addition and one for multiplication, which can execute in parallel. For highest performance it is therefore sensible to position arithmetic operations in the stream of instructions in a way that independent multiplications and additions can be scheduled for parallel execution.

Unfortunately the calculation for the update of the pdf is largely sequential. Listing 3.1 shows such a sequence. Relying on various precalculations of common sub-expressions it calculates the new particle distribution functions for *NorthWest* and *NorthEast* using *packed operations* where possible. The purpose is to illustrate that the additions and multiplications have to be executed in order. `addsd`, `subsd` and `addpd` are instructions that add or subtract the first and the second operand and store the result in the first one. `movlhps`, `movdqa` and `movhlps` are instructions that copy data from the second operand to the first. `mulpd` multiplies first and second operand and stores the result in the first one. Following the path of calculation one can see that in order to execute the first multiplication in line 7 it is necessary to finish the previous additions/subtractions, so it can't be executed in parallel. The next addition in line 10 is difficult to schedule in parallel to a multiplication, too, due to the direct dependency on operations in the previous two lines. The remaining additions require in-order execution as well and there are no further independent multiplications.

Listing 3.1: Excerpt of a version of the Lattice Boltzmann solver written in assembler for this thesis. The code snippet shows the cell update for the *NorthWest* and *NorthEast* direction

```
1  movsd xmm0, Ux_Register
2  addsd xmm0, Uy_Register
3  movlhps xmm0, xmm0
4  movsd xmm0, Uy_Register
5  subsd xmm0, Ux_Register
6  movdqa xmm2, xmm0
7  mulpd xmm0, three_Register            ; depends on line 5
8  mulpd xmm2, xmm2                      ; depends on line 6
9  mulpd xmm2, FourPointFive_Register    ; depends on previous line
10 addpd xmm0, xmm2                      ; depends on previous line
11 addpd xmm0, CommonCubicUTerm_Register ; depends on previous line
12 movlhps xmm2, Streamed_Cell_NorthEast_Register
13 movhlps xmm2, Streamed_Cell_NorthWest_Register_High
14 addpd xmm0, xmm2                      ; depends on previous line
15 movlpd Cell_NorthWest, xmm0
16 movhpd Cell_NorthEast, xmm0
```

Previous to the above discussed calculations the mass density and velocities must be calculated. As one can see in the definitions (2.1) and (2.2) it consists of additions and subtractions only, followed by one division per component of the velocity vector $\vec{u}$. There are no dependencies between the components otherwise, so *packed operations* can be used as well here. But still the multiplication unit is mostly idle during that calculation.

3.3 Theoretical Performance Bounds

With the ability to perform a multiplication and addition in parallel in one clock cycle an Athlon-64 for example with a clock frequency of 2.4 GHz has a theoretical peak performance P_{theo} of 4.8 GFlop/s. The same applies to Pentium 4/Xeon processors, with the capability of two Flops per cycle (SSE2).

A simple synthetic benchmark as in List. 3.2 that uses aggressive scheduling shows that it is actually possible to measure up to 94.7 % of this value on an Athlon-64 4000+, which is roughly 4.55 GFlop/s. So the upper limit reduces to the "technical" maximum P_{tech}.

Listing 3.2: Synthetic Maximum Flop/sBenchmark

```
1 .loop:
2 %rep 100
3         mulpd xmm0, xmm1
4         addpd xmm2, xmm3
5         mulpd xmm4, xmm5
6         addpd xmm6, xmm7
7 %endrep
8         dec ecx
9         jnz .loop
```

With regards to the Lattice Boltzmann algorithm such a sequence however cannot be achieved as there are more additions than multiplications. The minimum total amount of floating point operations per cell is approximately 156, with 90 additions/subtractions, 65 multiplications and one division.

The ratio of additions to multiplications reduces the approximation for the theoretically reachable maximum GFlop/srate. If n_{a} is the number of additions and n_{m} the number of multiplications, $1/2(n_{\text{a}} + n_{\text{m}})$ cycles are needed if two operations per cycle can be executed. However, when $n_{\text{a}} \neq n_{\text{m}}$, $\max(n_{\text{a}}; n_{\text{m}}) - \min(n_{\text{a}}; n_{\text{m}})$ additional cycles are required. As a result, the maximum reachable peak performance for the LBM P_{LBM} in comparison to P_{tech} is given by

$$P_{\text{LBM}} = \frac{1/2(n_{\text{a}} + n_{\text{m}})}{\underbrace{\min(n_{\text{a}}; n_{\text{m}})}_{\text{exec. at peak}} + \underbrace{\max(n_{\text{a}}; n_{\text{m}}) - \min(n_{\text{a}}; n_{\text{m}})}_{\text{additional instructions}}} \cdot P_{\text{tech}} \tag{3.1}$$

$$P_{\text{LBM}} = \frac{n_{\text{a}} + n_{\text{m}}}{2 \cdot \max(n_{\text{a}}; n_{\text{m}})} \cdot P_{\text{tech}}$$

That corresponds to 87 % of P_{tech}, which is approximately 4.0 GFlop/s on the analyzed machine.

With regards to the Lattice Boltzmann algorithm, performance is often measured in *M*ega *L*attice *S*ite *Up*dates per second, or like in this case for the performance of only fluid cells FluidMLSUPS. With the reduced GFlop/s rate an upper bound can also be

approximated in MLSUPS:

$$P_{\mathrm{LBM}} \approx 4.0 \text{ GFlop/s} = \frac{4.0 \text{ GFlop/s}}{156 \text{ Flops/cell}} \approx 26 \text{ FluidMLSUPS}$$

A more pessimistic boundary, based on the observations in the previous section that there are no calculations that permit parallel usage of the multiplication and the addition unit, would be to assume only 50 % of the peak performance, resulting in about 2.3 GFlop/s or 14 FluidMLSUPS.

4 Memory Performance Analysis

The purpose of this chapter is to give an overview over cache optimization techniques for use with the Lattice Boltzmann method. After an introduction on caches the 4-way blocking approach from [Igl03] is described, followed by another optimization step of fetching cell data ahead of actual use. In addition the possible influence of the virtual memory subsystem on the cache performance is investigated.

4.1 Cache Subsystems

Modern computer architectures feature small, hierarchically arranged buffers of memory between the main memory and the processor registers, known as *caches*. Their purpose is to reduce the average time to access main memory by providing high transfer rates and short access latencies. The levels of the hierarchy (*cache level*) towards the registers become smaller in size but also faster. For example an Athlon-64 has a level-1 cache for data with 64 kByte and a L2 cache with 512 or 1024 kByte (depending on the revision). If a requested piece of data is in the first level cache it can be accessed in average within 3 cycles, while access to the second level costs around 11 cycles, on an Athlon-64 for example. Main memory access on the other hand usually may take hundreds of cycles.

In general usage the caches are completely transparent to the programmer. There is no need to explicitly load the data to operate on into caches or evict back to main memory. But through changes in the data layout or the way an algorithm processes data it is possible to make caches visible. Tuning memory access patterns so that most accesses end up in cached memory makes the average latency drop significantly, causing the CPU to spend less time on waiting for data and therefore increase processing speed. This is particularly important for the Lattice Boltzmann algorithm in 3D, as it requires a lot of memory reads and writes.

Cache memory is divided into *cache lines*, usually 64 Byte each. Data from main memory to cache and between cache levels is transferred in whole cache lines only. This is very useful for example when reading from memory linear, but in case of the Lattice Boltzmann algorithm it may also mean that more data is transferred than immediately necessary. For example in *stream-collide* processing order 19 values are read from memory from very different locations. If for each value one cache line needs to be filled the demand for memory reads grows by a factor of 8 from $19 \cdot 8$ Bytes per double = 152 Byte to $19 \cdot 64$ Bytes per cacheline = 1216 Byte.

4.2 4-Way Blocking as a Cache Optimization Technique

There are different approaches to improve the re-use of data stored in the CPU caches. [Don04] presents a technique based on changes in the data layout. [Wil03] and [Igl03] instead change the way the grid is traversed, commonly called *cache blocking*. The best performing cache blocking from [Igl03] was used in this thesis, *4-way cache blocking*.

The basic idea is to divide the domain into little cubes, perform multiple timesteps inside one and move on to the next cube. The cubes are supposed to fit in the CPU caches. For easier understanding the following description is based on 2D, which corresponds to the 3-way blocking in [Wil03]. The extension to 3D is straightforward in code, merely an additional loop for the third dimension is necessary, with the same boundaries as for the other two dimensions.

The algorithm starts with an setup phase, where the boundaries are initialized. Figure 4.1 shows the pattern in which cells on the grid are updated. The example uses a 4^2 rectangle (block size) and also updates 4 times. Those two sizes are unrelated to the name *4-way blocking* though, which refers to the fact that all three dimensions in space as well as the time is blocked. The different colors show the amount of times a cell has been updated, the black rectangle is the area updated within one blocked time step. In this initial phase the update rectangle shrinks towards the boundaries of the grid.

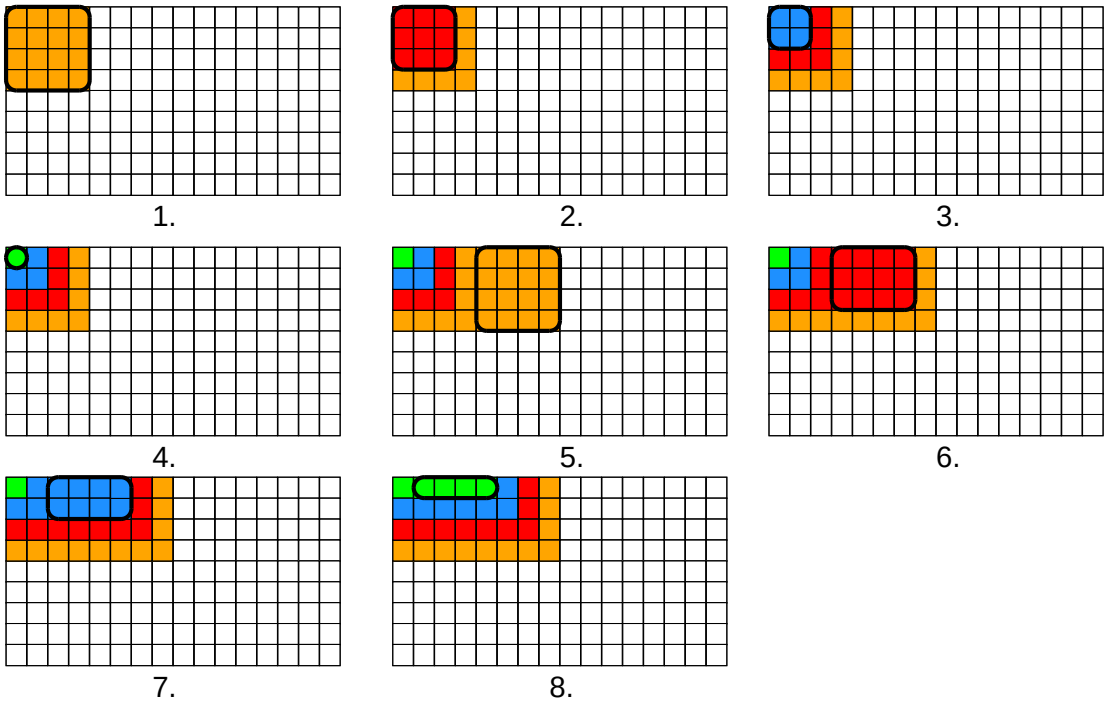


Figure 4.1: Startup Phase of 4-Way Blocking Update Pattern using 2D Illustration.

Once the startup phase is done the update rectangle keeps its size. As it shifts slightly

across the grid, cells are accessed multiple times, resulting in a lot of access to fast cache memory, lowering the average access time and thereby speeding up the whole algorithm. Figure 4.2 shows this main update phase with a total of eight time steps performed in two blocked iterations over the grid.

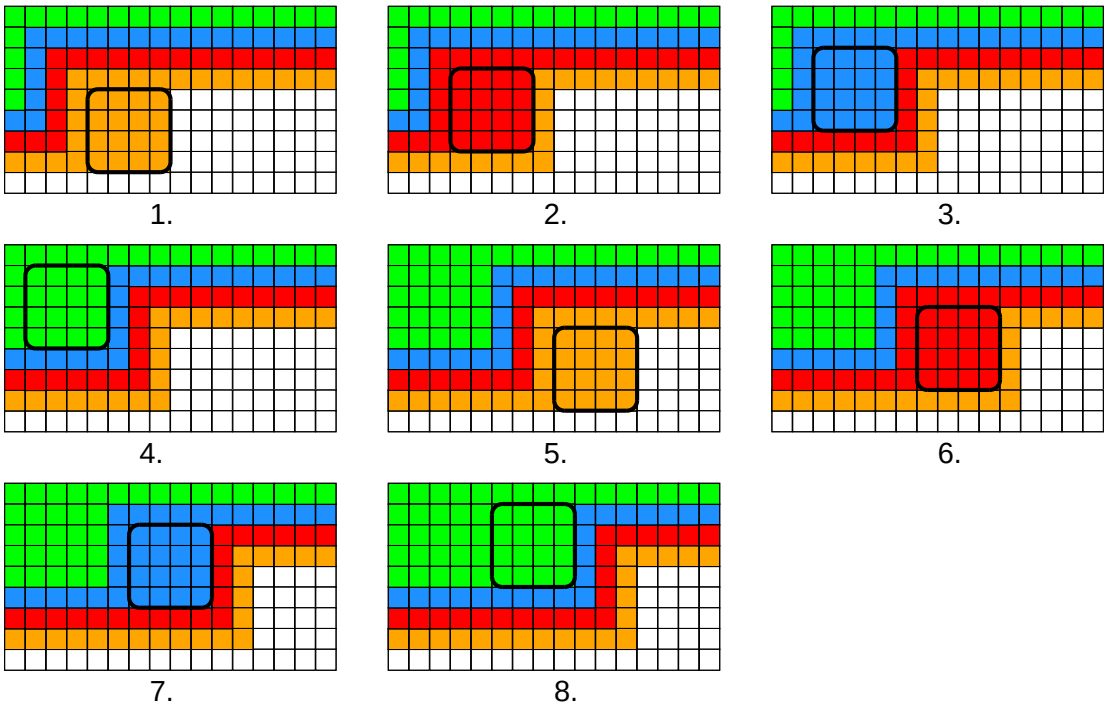


Figure 4.2: Main Phase of 4-Way Blocking Update Pattern using 2D Illustration.

Listing 4.1 shows the new loops for the 4-way blocking. Of particular interest are the `Z/Y/X_Start` and `Z/Y/X_End` macros, which make sure that the `x/y/z` variables stay within the boundaries of the grid, if the grid dimensions are not exactly divisible by the block size. In addition they implement the shifting of the update rectangle during the inner timesteps. Listing 4.2 shows the macro definitions for the x dimension.

Listing 4.1: Skeleton of the 4-Way Cache Blocking for 3D

```
// divide domain into cubes
for (zz = 1; zz < GridDepth - 1; zz += BlockSize)
    for (yy = 1; yy < GridHeight - 1; yy += BlockSize)
        for (xx = 1; xx < GridWidth - 1; xx += BlockSize)

            // process each block multiple times
            for (t = 0; t < TimeBlock; ++t) {

                for (z = Z_Start; z < Z_End; ++z)
                    for (y = Y_Start; y < Y_End; ++y)
                        for (x = X_Start; x < X_End; ++x) {
                            // main lattice boltzmann cell step
                            ...
                        }

                // swap source and destination grids
                ...
            }
```

Listing 4.2: Definition of the X_Start and X_End macros

```
#define X_Start Max(1, xx - t)
#define X_End (((xx + BlockSize) < (GridWidth - 1)) ? \
               (xx + BlockSize - t) : (GridWidth - 1))
```

The blocking in time may present a problem for an actual application of the Lattice Boltzmann method though. The combination of multiple time steps into one iteration for examples makes it impossible to *see* the state of the whole fluid in the intermediate steps. That not only makes visualization difficult but may also make for example simulations that combine the Lattice Boltzmann method with other physical simulations in one program impossible when data in the intermediate timesteps is needed.

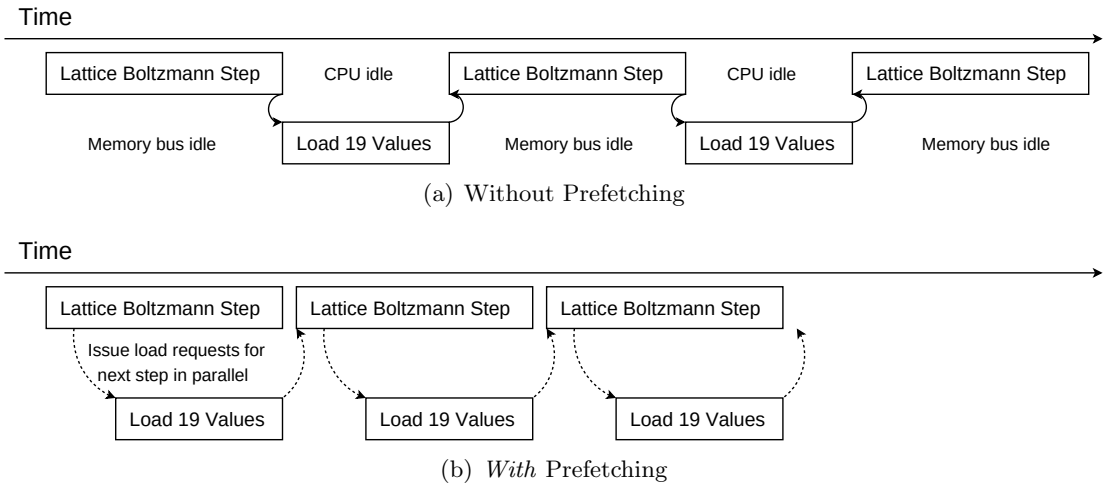
4.3 Software Prefetching

The Lattice Boltzmann algorithm has two distinct patterns of accessing memory. The streaming step causes values to be gathered or pushed from the neighboring cells. What appears to be a direct neighborhood in a grid by just an increment or decrement of the x, y or z coordinate (see `Cell_<Direction>` and `New_Cell_<Direction>` macros in List. 2.4 and 2.5) may result in a big distance from the current cell in the actual linear addressed memory. On the other hand reading the current 19 cell values (*collide-stream*) or writing them (*stream-collide*) results in a linear memory access pattern, as the cells are processed in order. So in the algorithm either the cell read operations are scattered and the results are written in linear fashion or the other way around.

[Cor04a] and [Dev04] explain that modern Intel and AMD processors attempt to detect regular read access patterns and then start loading memory further ahead into cachelines, the *hardware prefetch mechanism*. The idea is to fetch the data needed in the next loop iteration while the processor is busy with arithmetic operations in the

current one. In stream-collide order in the Lattice Boltzmann algorithm the scattered read operations are likely to prevent the CPU from seeing a pattern simple enough for the hardware prefetcher. In fact for Athlon-64 and Opteron processors it is documented that the mechanism only starts prefetching subsequent cache lines if there are linear cache line wise memory reads.

In that case [Cor04a] and [Dev04] instead suggest the use of software prefetch instructions, to reduce the average latency of memory access. With regards to the Lattice Boltzmann algorithm in stream-collide order the goal is to achieve a similar memory bus utilization as with sequential reads in collide-stream but retain the linear writing pattern for the results (see also Fig. 4.3).



(a) Without Prefetching

(b) *With* Prefetching

Figure 4.3: Simplified Model of Execution Path and Memory Bus Utilization

4.4 Large Page Optimization

Modern architectures divide the logical address space a program can use into small units, *pages*. A common size for a page is 4096 Byte. For each page an entry in a structure exists that defines exactly to which equally sized piece of main memory it is mapped. That structure in turn is part of a hierarchy of lookup tables called *page tables* and *page directories*, managed by the operating system. This level of indirection in memory access is used for example to implement virtual memory.

Mapping an address in virtual memory to a physical address is an expensive operation as it involves lookups in the page tables held in main memory. In order to avoid such a lookup for every memory access an on-chip cache exists that holds a list of last recently used virtual addresses and their corresponding physical addresses, the so-called *Translation Lookaside Buffer* (TLB). Modern processors even have multiple TLBs, one for each cache level.

The relation between the TLB and the cache is founded in the process of verifying that

a requested piece of data can successfully be served from a memory cache (*cache hit*). First a cache line is searched that corresponds to the given requested virtual address. But then it is still necessary to verify that the actual physical memory the cache line holds is from the same address than the requested virtual address points to. That is why it is necessary to perform the translation, preferably through a fast lookup in the TLB.

The size of pages in virtual memory management directly affects the efficiency of the TLB cache through the amount of address space it can cache. On the Athlon-64 and Opteron for example the level 2 TLB has 512 entries. With a page size of 4096 Byte by default this multiplies to an address space of only 2 MByte.

Large domains with the Lattice Boltzmann method in 3D require a lot of memory and as such a lot of pages of virtual memory. Therefore the influence of the TLB on the run-time performance grows as the domain grows.

On the other hand processor architectures often support multiple page sizes, 2 or 4 MByte are common additional sizes. Hence with a bigger page size the number of expensive lookups decreases.

5 Results

In this chapter measurements on three different machines are presented. The different (Fluid)MLSUPS rates for arithmetic, in-cache and in-memory performance are put into relation. The machines shown in Tab. 5.1 were used.

Table 5.1: Architectures used for testing.

	Athlon-64 4000+	**Xeon Nocona**	**Opteron 248/848**
clock frequency / GHz	2.4	3.4	2.2
Dual Channel RAM	DDR400 6.4 GByte/s	DDR333 5.3 GByte/s	DDR333 5.3 GByte/s
L1 instr. cache / kByte	64	12[1]	64
L1 data cache / kByte	64	16	64
L2 cache / kByte	1024	1024	1024
cache line length / Byte	64	64 (128[2])	64
TLB entries	L1 instr. and data: 32 per 4 kByte page 8 per 2 MByte page L2 data: 512	Data TLB: 64 Instr. TLB: 64	like the Athlon-64
Measured Memory Transfer Rate / GByte/s	6 read write and copy	6.1 read 4.5 write 3.7 copy	4.7 read 4.6 write 4.5 copy
L2-L1 Transfer Rate GByte/s	12 read 11 write	53 read 16 write	11 read 10 write
L1 Transfer Rate GByte/s	18 read and write	49 read 12 write	16 read and write
L1 Cache Latency (cycles) L2 Cache Latency (cycles)	3 11	3 27	3 11

[1] The instruction cache in Pentium based processors stores μ-operations.
[2] Always two cache lines are loaded from memory.

5.1 In-Cache Results

As a first step for comparison the core LBM calculation sequence was extracted from the solver and on assembler level all instructions that result in memory access were removed. The remaining code of course does not calculate an actual fluid flow, but it consists of exactly the calculation sequence for doing so, minus memory access though.

This version was compared to the complete fluid solver in *stream-collide* order with domain sizes where all allocated memory fits into at least the level 2 cache, with a size of 1 Megabyte on all machines. Another *important* property was that the domain consisted entirely of fluid cells, as we are interested in comparing the arithmetic performance of the fluid flow calculation, not how much time it takes for the processor to move data around with the no-slip boundary condition handling.

Figure 5.1 shows the results on the Athlon-64 4000+. The results indicate a fairly constant performance inside the cache, except when the grid size grows near to the total size of the L2 cache. A size of $14^3 \cdot 19$ (values per cell) $\cdot\, 2$ (grids) $\cdot\, 8$ (bytes per double) fills already 83 % of the L2 cache. Given how near the *in-cache* unblocked plain solver is to the pure arithmetic version it appears that the caches can deliver data reasonably fast on that machine. However the measured ≈ 8 FluidMLSUPS are quite in distance to what the processor could achieve in theory. As discussed in Sec. 3.3 the processor itself may be able to produce something in the range of 14 to 26 FluidMLSUPS, depending on how good the instruction scheduler can feed the floating point units.

As with the Athlon-64 the Xeon Nocona shows a reasonably constant *in-cache* performance, too (Fig. 5.2). The drop of performance towards the end of the L2 cache is not that visible though. A difference to the Athlon-64 however is the significant distance between the measured arithmetic fluid update performance and the *in-cache* performance, almost 2 MLSUPS. One possible reason for this difference is the higher latency of the Xeon's L2 cache, it may make stalls in the long pipeline of the Pentium core more expensive than on the Athlon-64.

With a clock frequency of 3.4 GHz one would expect a better performance of the code without memory access. However the benchmark of listing 3.2 achieved only 3.3 GFlop/s, which is only half of the theoretical peak of 6.8 GFlop/s, executing one multiplication and one addition in parallel per cycle (2 FLOP/cycle). With the multiplication/add ratio from Sec. 3.3 87 % of 3.3 GFlop/s correspond to approximately 13 FluidMLSUPS, about twice as much as the measured 7 FluidMLSUPS.

At last Fig. 5.3 shows the results on the AMD Opteron. The processor has the same core as the Athlon-64 and the graphs look similar. One difference is the lower overall performance, which appears to be due to the fact that the Opteron has a lower clock frequency. Also the attached memory is of a slower type, which explains the bigger drop off performance towards the end. That's where main memory effects become visible.

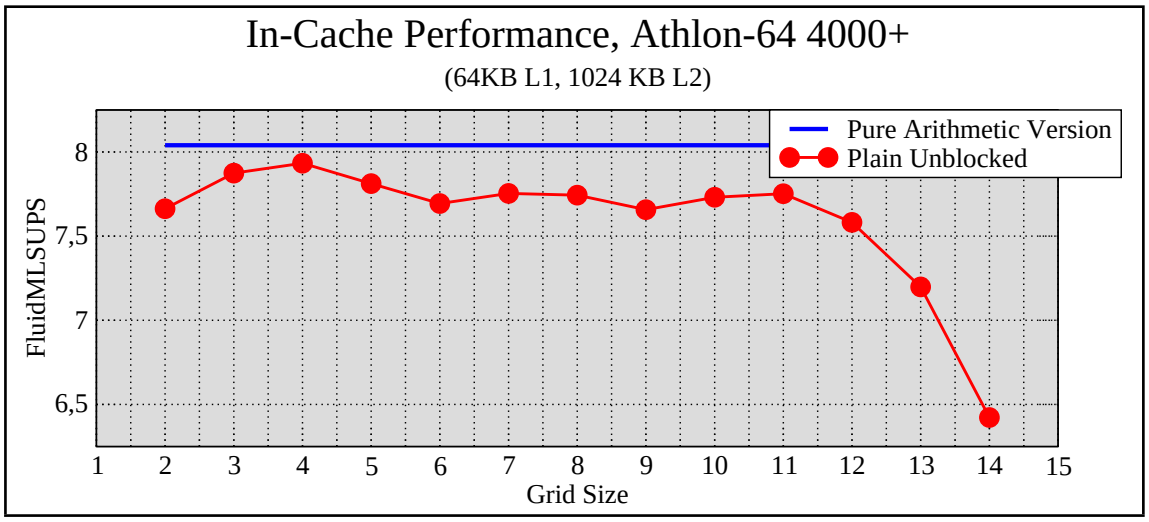


Figure 5.1: Comparison of in-cache version against measured arithmetic performance on the Athlon-64 4000+.

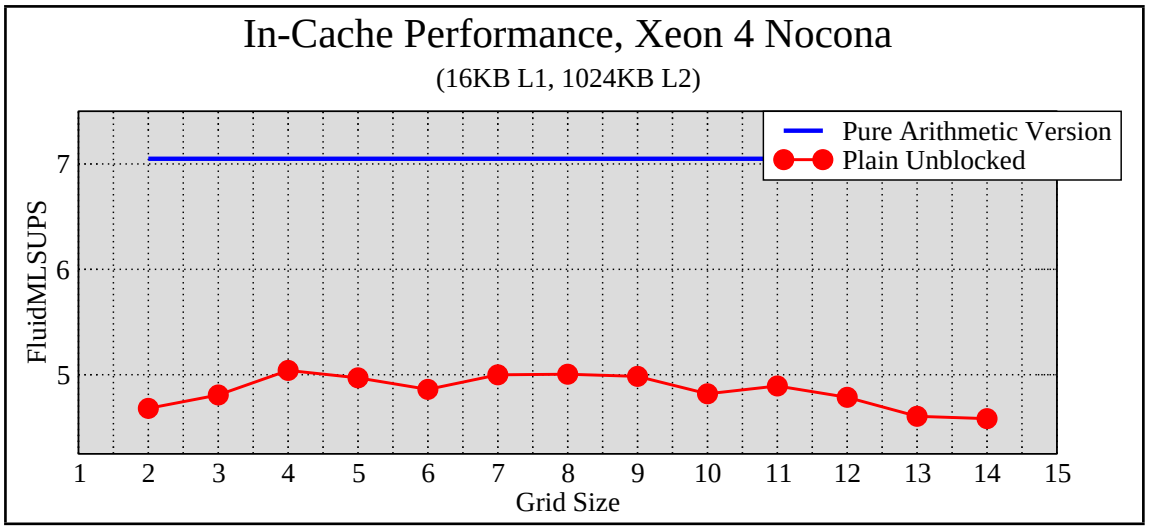


Figure 5.2: Comparison of in-cache version against measured arithmetic performance on the Xeon Nocona.

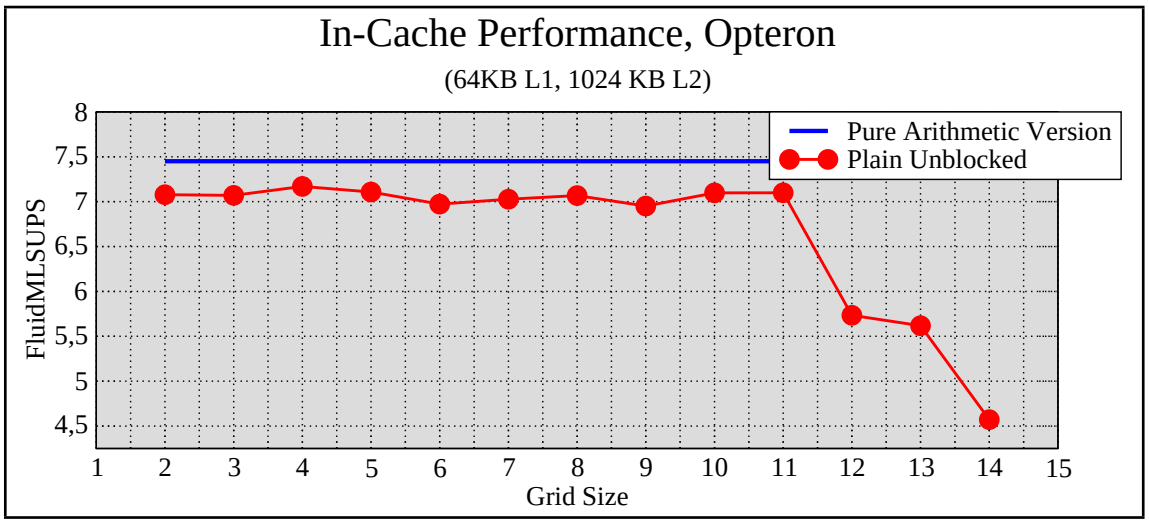


Figure 5.3: Comparison of in-cache version against measured arithmetic performance on the Opteron.

5.2 Memory Results

In this section the results of the cache optimizations are shown, the 4-way blocking as well as the software prefetching on top of it. As with the in-cache measurements in the previous section the domain consisted of fluid cells only and stream-collide order was chosen, as it turned out to be faster than collide-stream in this case.

Figure 5.4 shows the results on the Athlon-64 4000+. As already shown in [Igl03] the 4-way blocking provides a significant speed-up over the unblocked version. The software prefetching cuts execution time in average by 15 %, resulting in one more MLSUPS. However the 4-way blocked version of [Igl03] with a compressed grid and a block size of 8^3 still provides the best performance.

The key insight becomes apparent when comparing these results with the average in-cache performance: The block techniques effectively reduce the high latency of main memory, $\approx 80 - 90$ % of the fluid in-cache performance is achieved on this machine.

The measurements on the Xeon Nocona are shown in Fig. 5.5. Again an improvement of the 4-way blocking can be seen, however the difference to the unblocked version is slightly less than on the Athlon-64. The software prefetching on top of the blocked code brought only little improvement, but it's still visible. But again the cache optimized code is at approximately 84 % of the in-cache results, showing the efficiency of the optimization.

The measurement results on the Opteron machine as in Fig. 5.6 draw a similar picture as on the Athlon-64. However the difference between the fasted cache optimized version and the average *in-cache* performance is slightly bigger, most likely due to the slower memory (DDR333 versus DDR400 on the Athlon-64). The 4-way blocked code with software prefetching is at approximately 79 % of the in-cache average.

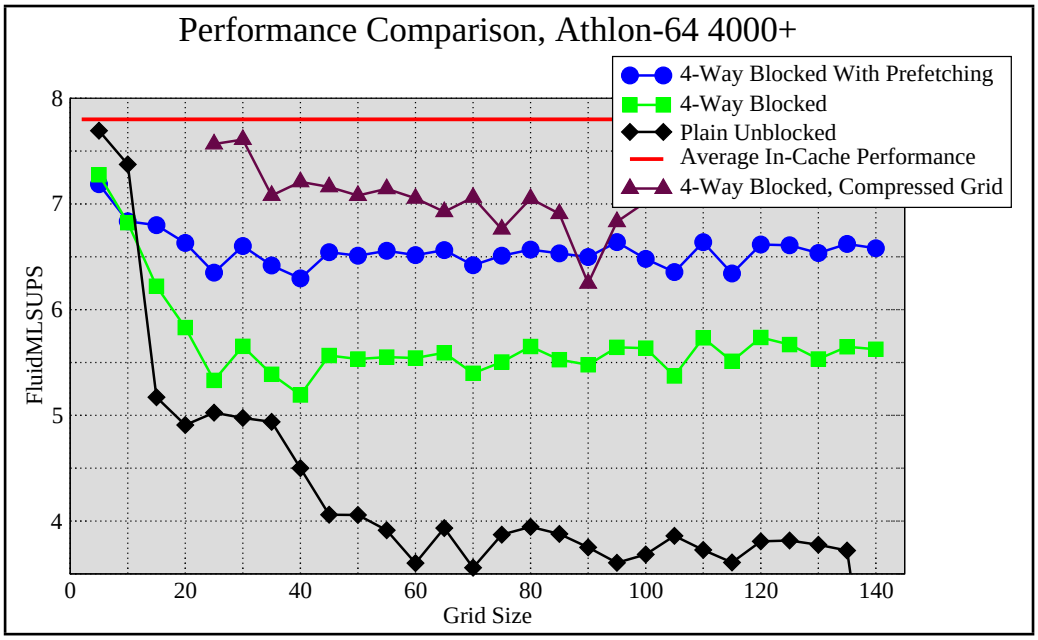


Figure 5.4: Comparison of cache blocking techniques with *in-cache* performance on the Athlon-64 4000+.

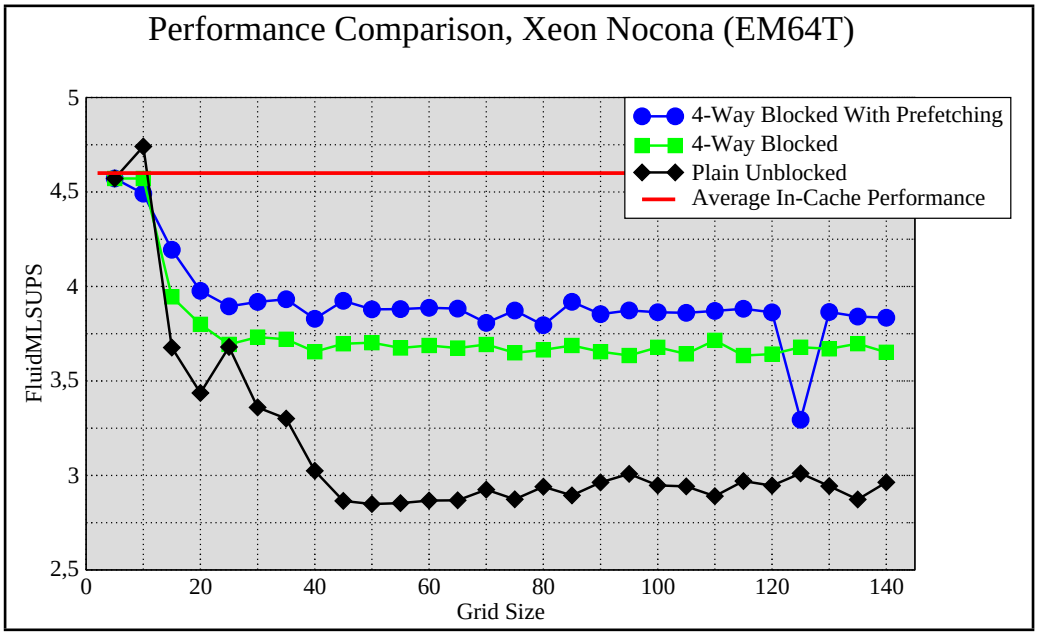


Figure 5.5: Comparison of cache blocking techniques with *in-cache* performance on the Xeon Nocona.

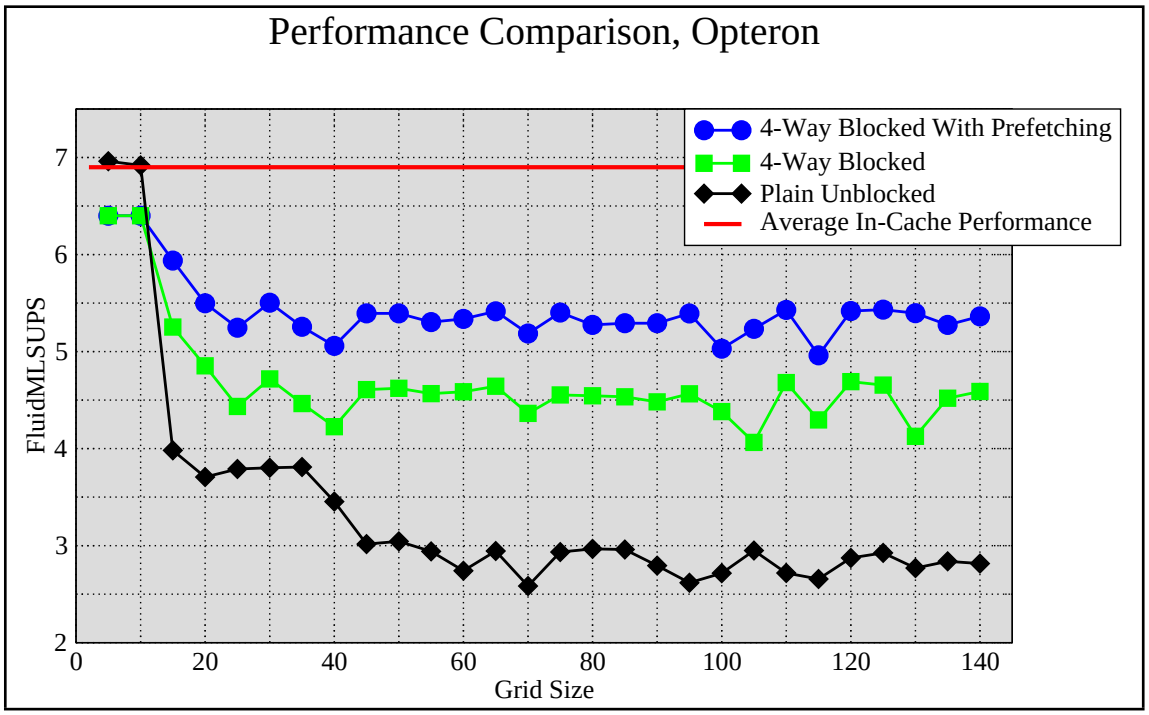


Figure 5.6: Comparison of cache blocking techniques with *in-cache* performance on the Opteron.

5.2.1 Large Pages

The effect of large pages on the overall performance was measured on the Opteron machine. The 4-way blocked code with prefetching was run once with the system default page size of 4096 Bytes and then using a *pre-loaded* library developed for this thesis that ensures the allocated memory for the grids end up in reserved memory areas for which the operating system guarantees a bigger page size, 2 MByte on the Opteron/Athlon-64.

The results of the performance measurements with large pages are shown in Fig. 5.7. In parallel to that the sum of the Data TLB misses per cell for both cache levels were measured (see Fig. 5.8), using the *on-chip* hardware performance counters. It should be noted that what appears like a big fluctuation in the performance is not actually significant in actual MLSUPS. On some grid sizes a huge increase in the demand for different pages can be seen in the graph for the TLB misses with 4 kByte pages. Without deeper analysis of the page access patterns of the already quite complex 4-way blocked code one can only speculate about the reasons for this behavior. However it is visible that in average the use of large pages did improve the performance by a small amount.

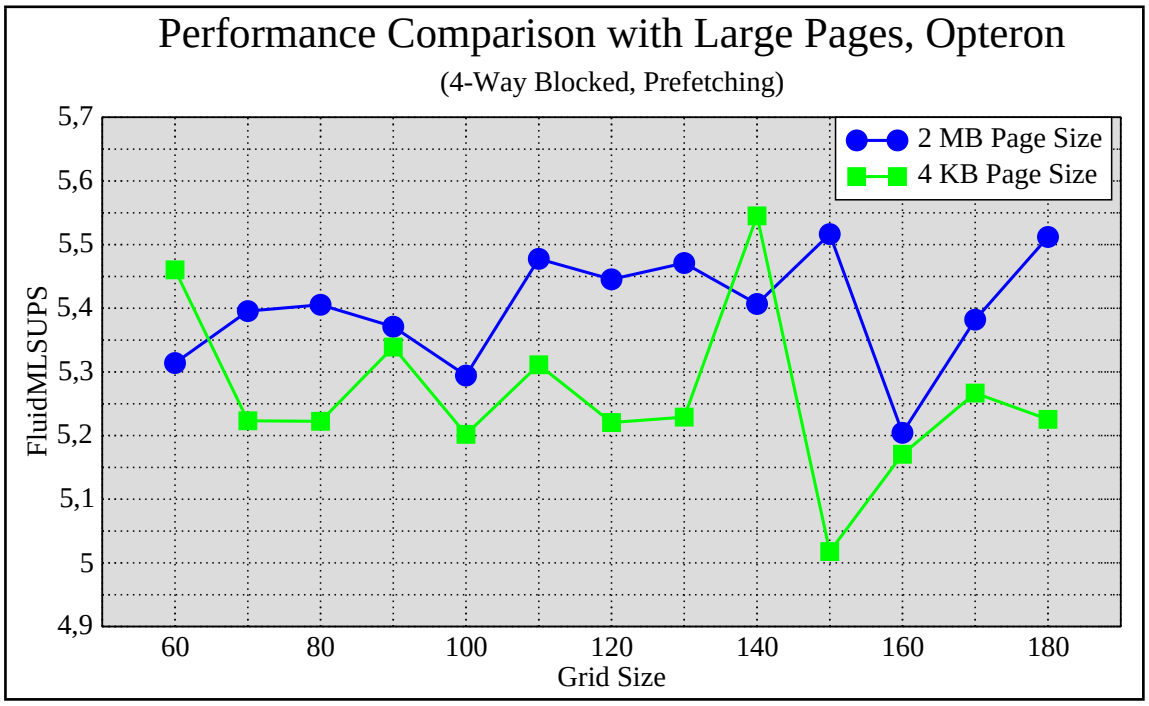


Figure 5.7: Comparison of performance with different page sizes on the Opteron.

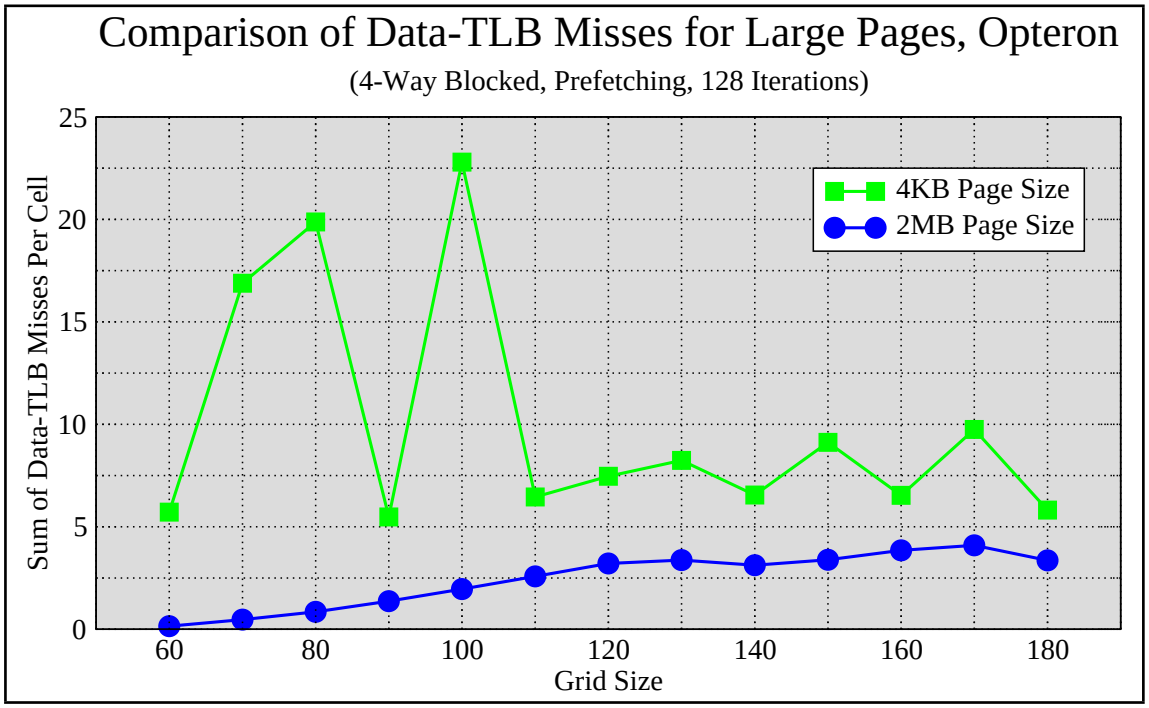


Figure 5.8: Comparison of combined L1 and L2 Data-TLB misses per lattice cell for small and large pages on the Opteron.

6 Conclusion

The D3Q19 Lattice Boltzmann method is very demanding in terms of computation and memory usage.

A lot of effort has been spent to decrease memory consumption and computing times, such as 4-way blocking [Igl03], compressed grid [Wil03] or different data layouts [Don04].

Dependencies in the calculation make it difficult for the processor's instruction scheduler to execute operations out-of-order. Also the unbalanced ratio of multiplications to additions prevent a maximum load on separate *add* and *mul* floating point execution units.

The task of this thesis was to examine x86-64 architecture specific optimization techniques for use with the Lattice Boltzmann method. Experiments showed that current compilers were unable to generate code for this algorithm that made use of *packed* SSE2 operations. Also the insertion of *software prefetch* instructions using inline assembler statements is not very reliable, because one cannot be sure how exactly the code in the high-level language before or afterwards is mapped to actual instructions and whether these result in memory access to the cache line that is supposed to be filled in advance. That is why an implementation of the algorithm was written entirely in assembler.

Under theoretical consideration the algorithm is limited by the data rate the memory can deliver, as shown in [Don04]. However comparing the efficiency of the cache optimized code with the speed obtained with *in-cache* fitting fluid domains produced interesting results. In combination with *software prefetching* the 4-way cache blocked code was able to achieve 80-90 % of the FluidMLSUPS rates measured in the L2 cache. This indicates that the cache optimization was efficient enough to hide large portions of the high latency and slow data rate of the main memory.

On the Athlon-64 4000+ 6.5 FluidMLSUPS were measured for grids that certainly exceed the CPU cache sizes. In the worst case $19 * 8 = 152$ Bytes are written and $19 * 64$ Bytes per cacheline $= 1216$ Bytes are read. That would correspond to a theoretical memory transfer rate of $(1216 + 152)$ Bytes $* 6.5$ FluidMLSUPS $= 8.8$ GByte/s. Obviously that is not the actually achieved rate as the 6.5 FluidMLSUPS were measured with 4-way cache blocking, for which the goal is to keep the slightly shifting cube inside the L2 cache for all blocked time steps. With four time steps in one iteration one could approximate the real transfer rate with $\frac{1}{4} \cdot 8.8 = 2.5$ GByte/s. That approximation in turn indicates that the memory bus is not entirely loaded, given that 6 GByte/s were achieved for memory transfers in benchmarks on that machine.

In the next step the performance results from the *in-cache* measurements were put into relation with a version of the code that had all memory access instructions removed,

in order to see how much the latency of the caches influence the execution time. The comparison on the Athlon-64 4000+ and on the Opteron showed little difference in performance, the caches did not appear to be a limitation. On the Xeon Nocona a slight drop was measured though.

Given the efficiency of the cache optimization and the comparison of *in-cache* to "pure arithmetic" it appears the remaining limiting factor is somewhere in the path from instruction decoding through scheduling up to the floating point execution units. For an estimation the raw floating point processing power can be reduced due to an unbalance of multiplications to additions (section 3.3). When comparing that with the measured performance a big gap remains. Less than 50 % of the estimated peak performance was reached, on all three platforms.

It remains to be determined which factors exactly in the coding of the Lattice Boltzmann algorithm have the biggest impact on the arithmetic performance. Despite the out-of-order execution capabilities of Athlon and Pentium processors the actual placement of instructions in the machine code remains to have a big influence on the performance. Compilers tend to be good in instruction scheduling but appear to have problems in transforming the complex equations to *packed* instructions. The combination of both may possibly improve performance even more.

List of Tables

Listings

List of Figures

A Terminology on Assembler Listings

Several assembler source listings are shown in this thesis and the purpose of this appendix chapter is to explain the expressions used.

A common property of all the commands shown is that generally assignments are read from right to left, with the value of the right argument being assigned to the left argument. This notation is also known as the *Intel Style*. Listing A.1 shows some very basic operations for loading values into registers.

Listing A.1: Register Assignments

```
1 mov eax, 42              ; load integer constant 42 into 32-bit eax
2                          ; register
3 mov ebx, [SomeAddress]   ; load value from memory at SomeAddress
4 mov ecx, edx             ; copy value from edx register to ecx register
5 mov eax, [ebx]           ; interprets the value of the ebx register
6                          ; as address and loads the 4-byte value from
7                          ; that address into eax
```

Table A shows a list of common arithmetic operations used, together with their assembler abbreviations and their argument usage.

Table A.1: Common Arithmetic x86 Assembler Operations

Abbreviation	Description or equivalent C code
`add destination, source`	integer addition: destination += source
`sub destination, source`	integer subtraction: destination -= source
`inc arg`	arg += 1
`dec arg`	arg -= 1

Another pattern is the combination of an integer arithmetic operation with a branch to a label, depending on the previous result. Listing A.2 demonstrates a simple loop that is executed 100 times.

Listing A.2: Simple Loop

```
1     mov ecx, 100 ; use ecx as counter register
2                  ; and initialize it with '100'
3 .someLoopLabel:
4     ...
5     dec ecx              ; subtract one off ecx. if the result is
6                          ; zero the so-called zero bit flag is set,
7                          ; otherwise it's cleared.
8     jnz .someLoopLabel   ; jump-if-not-zero, branches to .someLoopLabel
9                          ; if the zero flag is not set.
```

B Memory and Cache Benchmark Tool

The initial motivation for the development of this benchmark tool were simple memory copying tests that showed surprisingly different performance on otherwise supposedly similar machines, even though the code was trivial. From these experiments the desire grew to learn more about the different techniques for getting the most out of a given architecture in terms of memory/cache operations. Knowing the memory and cache performance limits of a given machine makes it much easier to compare real application code, it allows to see how much there is left to tweak in terms of performance. At the time of writing of this thesis the tool is capable of measuring various aspects of the memory and cache subsystem:

- Data rate for pure reading and writing to main memory, as well as the combination by memory copying.
- Data rate between L1 and L2 cache
- Data rate between L1 and CPU
- Average latency for read operations from the caches
- Effective size of cache lines

In order to provide a high level of accuracy it was necessary to write the actual benchmark routines in assembler. That's particularly important for timing sensitive measurements like the cache line size detection. That is why the program is limited to the x86 processor family.

B.1 Memory Rate Benchmarks

The purpose of memory rate benchmarks is to measure the maximal throughput the combination of processor, memory controller, bus and ram chips can provide.

For the reading data-rate a block of memory with a fixed size is transferred from memory to the CPU cache or even a register. The size exceeds the largest CPU cache. Analogously the same happens for writing and copying.

Experiments have shown that different methods lead to the maximum achievable data transfer rate. For example some machines appear to prefer SSE2 based 16-byte wise memory operations, while others get the highest throughput with 8-byte wise MMX operations. That is why a broad range of methods using various techniques are implemented.

B.1.1 Memory Reading Techniques

Cache Line Wise Memory Reads

One approach to quickly feed data into the cache is to exploit the fact that the processor always loads whole cache lines from memory, usually 64 bytes wide. This allows a more efficient use of the limited amount of load instructions that can be scheduled, by issuing memory load requests with an address distance equal to the cache line size. Listing B.1 demonstrates the idea.

Listing B.1: Example of cache line wise memory reads

```
...
mov eax, [BaseAddress]  ; trigger load of cacheline at BaseAddress
                        ; by requesting the first 4 bytes to be moved
                        ; into the eax register

mov eax, [BaseAddress + 64] ; trigger a load 64 bytes further, so that
                            ; another cache line gets loaded
mov eax, [BaseAddress + 128]
mov eax, [BaseAddress + 192]

; At this point 4 * 64 = 256 bytes have been loaded into the cache
; although only 4 * 4 = 16 bytes have reached a register

...
```

Memory Reads Using Software Prefetching

Another approach that turned out to perform remarkably well on the Pentium 4 Prescott is to use software prefetch instructions within the actual loading code. They initiate read requests to load the entire cache line containing a specified address. Listing B.2 shows an excerpt for such a combined read and prefetch sequence.

Listing B.2: Example of MMX based memory reads mixed with software prefetching instructions

```

    mov esi, BaseAddress          ; load base address into esi register
.loop:
    prefetchnta [esi + 1024]      ; request the cache line 1024 bytes
                                  ; ahead to be loaded

    movdq mm0, [esi]              ; load the first 8 bytes into mm0
                                  ; register

    movdq mm1, [esi + 8]          ; load the next 8 bytes

    prefetchnta [esi + 1024 + 16] ; issue another prefetch request to
                                  ; make sure cache lines are loaded
                                  ; in the next loop run

    movdq mm0, [esi + 16]
```

```
17      movdq mm1, [esi + 24]
18      prefetchnta [esi + 1024 + 24]
19      ...
20
21      add esi, 1024                   ; 1024 bytes have been loaded, so
22                                      ; advance the base address pointer
23
24      dec ecx                         ; decrement loop counter and
25      jnz .loop                       ; and jump to loop start
```

Memory Reads Using Block Prefetching

On an Athlon 64 a slightly different method gave the best performance. The basic idea is to split the total data set into blocks with a fixed size. Then before operating on one block it is explicitly loaded into cache with a separate dedicated loop. This fetching in advance can be done using direct moves into processor registers or by using the software prefetch instructions.

Listing B.3: Example of SSE2 based memory reads using separate block prefetching

```
1
2   %define BlockSize 16384  ; let the size of the block to operate
3                            ; on be 16384 bytes
4
5       mov edx, TotalDataSize ; load the total amount of data
6                              ; we want to process into the edx
7                              ; register
8
9       mov ebx, BaseAddress ; load base address into ebx register
10  .mainloop:
11
12      ; start loading the working block in 1024 byte chunks
13      ; into the cache
14
15      mov esi, ebx           ; keep the current block start address
16                             ; in ebx and copy it to esi to use it
17                             ; in the prefetch loop
18      mov ecx, BlockSize     ; load block size into ecx register,
19                             ; working as counter of the number of
20                             ; bytes that are left to prefetch
21  .prefetchloop:
22      prefetchnta [esi]
23      prefetchnta [esi + 64]
24      prefetchnta [esi + 128]
25      ...
26      prefetchnta [esi + 1024 - 64]
27
28      add esi, 1024          ; advance our pointer inside the block
29                             ; we prefetch
30
31      sub ecx, BlockSize
32      jnz .prefetchloop      ; jump back to the .prefetchloop label
33                             ; if the result of the previous subtraction
34                             ; is non-zero (jnz), that is if there
```

```
35                          ; are bytes left to prefetch
36
37      ; now that the block is in the cache we can work on it. in
38      ; this case by reading its content into registers, for measuring
39      ; the time it takes
40
41      movdqa xmm0, [ebx]        ; load data from cache into sse2
42      movdqa xmm1, [ebx + 16]   ; CPU registers
43      movdqa xmm2, [ebx + 32]
44      movdqa xmm3, [ebx + 64]
45      ...
46
47      add ebx, BlockSize    ; advance our base address pointer
48
49      sub edx, BlockSize    ; reduce the counter for the amount of
50                            ; bytes that are left to process.
51      jnz .mainloop         ; unless it drops to zero, as flagged by
52                            ; the previous subtraction continue by
53                            ; branching to the top .mainloop label
```

B.1.2 Memory Writing Techniques

For memory writing the fastest way to write to memory, just as recommended in [Dev04] and [Cor04a], is to use the so-called *non-temporal* store instructions. They appear like regular data copying instructions, but their usage indicates that data is written without the need to reside in the cache for soon reuse and therefore permit writing directly to memory without first having to load a cache line.

B.2 Cache Benchmarks

The cache benchmarks attempt to discover the limits of bandwidth and the average access latency of the cache subsystem.

Measurements in the caches are very timing sensitive due to their small sizes. Only small blocks of data can be used for reading and writing, only few actual instructions get executed. Reliable results require the use of the processor's time stamp counter which is automatically incremented every clock cycle. When counting the amount of cycles a benchmark takes it is important to keep in mind that modern processors may execute the instructions out of order. In order to avoid the situation where the instruction for reading the time stamp counter value happens to get scheduled before the execution of an instruction relevant for the benchmark it is necessary to finish with a serializing instruction that flushes the pipeline completely. However this instruction itself can take a considerable amount of cycles, too. So in addition it is necessary to measure the amount of cycles that serializing instruction takes, in a separate procedure, and subtract it from the amount of cycles measured in the actual benchmarking routine.

B.2.1 Cache Data Rate Benchmarks

As with the memory benchmarks the cache data rate test reads or writes a block of memory. For the rate between the L1 and the L2 cache it is sufficient to use simple cache line wise loads to trigger the transfer to the first level. Essentially listing B.1 can be re-used for that. The block used must be completely in the second level though and of course it has to be bigger than what the first level can just hold.

For the rate between the L1 cache and the CPU registers it is again important that the transferred block fits entirely into the first level cache and that no cache line wise transfers are used but real ones instead, ranging from 4 to 16 bytes (largest register size available).

B.2.2 Cache Latency Measurements

Essentially the aim is to measure the amount of cycles a single load instruction from the cache of interest to a register takes.

However counting the cycles of just one such instruction does not give stable results. So it is necessary to execute multiple ones, exactly in order. Iterating through a list of linked pointers where each pointer holds the address of the next imposes a dependency that ensures the in-order execution, and that one load is finished before the next one is started.

Making that linked list circular by letting the last entry point to the first one again permits executing even more instructions by traversing the whole list multiple times.

Using this approach produced quite stable results, but the measured cycles were still quite higher than one would expect. We were not able to discover the exact reason, but it turned out that introducing independent dummy instructions between two depending loads influenced the results greatly. An example for such an instruction would be an integer addition of two completely unrelated registers. Different amounts of such unrelated instructions were required to reach a minimum measured latency. What gave the shortest access time on a Pentium 4 not necessarily gave the best on an Athlon64, too. The amount of cycles these extra instructions take needs to be measured in separate and subtracted from the total result.

Also, for measurements in the first level cache it is important that the complete buffer fits exactly into it.

For measurements in the second level cache it is important to avoid that the data read is already in the first level cache, as we *only* want to see the second level. It's hard to say when exactly that could happen though. Experiments showed sensible results when choosing a distance between two consecutive pointers in the linked list that is equal to the size of the first level cache.

B.2.3 Detection of Effective Cache Line Size

The size of cache lines is an important property of a cache level. Even though it is generally documented it may be useful to be able to also measure it, in particular to discover architectures like the Pentium 4 which for the second level cache always loads two 64 byte lines, giving effectively 128 byte lines.

The approach is based on the latency measurement technique from section B.2.2. Again the total amount of cycles of consecutive memory reads from a circular linked pointer list is measured. The main difference is a change in the layout of the linked pointer list. The buffer is split up into segments, with each being sized so that multiple cache lines would fit into one segment. Within each segment two loads are performed, the first one at the beginning of the segment and the second one at an increasing distance. Figure B.1 illustrates the layout and the procedure.

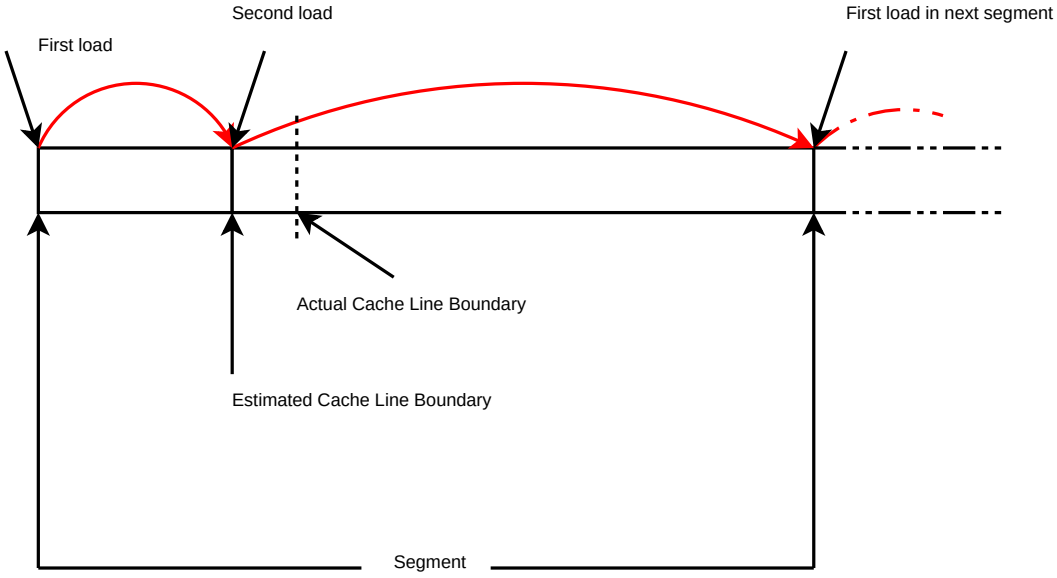


Figure B.1: Illustration to Cache Line Size Detection

If the second load within the segment is within the first cache line then it takes up less cycles than when it hits the cache line boundary and therefore requires a second cache line. This change in cycles is visible, as figure B.2 shows and its position with regard to the used distance indicates the effective line size.

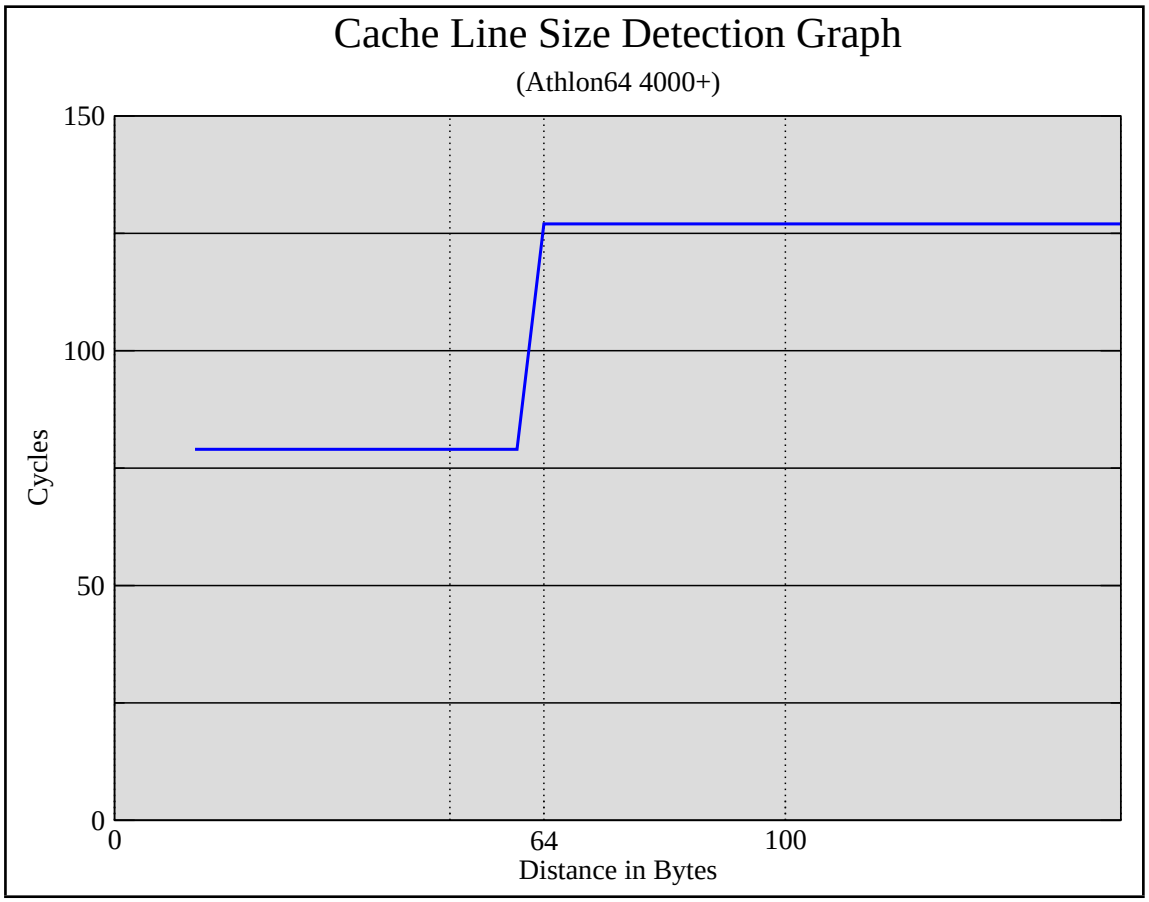


Figure B.2: Cycles for Level 2 Cache Line Size Detection on an Athlon-64 4000+.

Bibliography

[Cor04a] Intel Corporation. IA-32 Intel Architecture Optimization Reference Manual, 2004. ftp://download.intel.com/design/Pentium4/manuals/24896611.pdf . 22, 23, 46

[Cor04b] Intel Corporation. Intel Extended Memory 64 Technology Software Developer's Guide, June 2004. http://www.intel.com/technology/64bitextensions/30083402.pdf http://www.intel.com/technology/64bitextensions/30083502.pdf. 13

[Dev03] Advanced Micro Devices. AMD64 Architecture Programmer's Manual Volume 1: Application Programming, September 2003. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf . 13

[Dev04] Advanced Micro Devices. Software Optimization Guide for AMD Athlon(tm) and AMD Opteron(tm) Processors, November 2004. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF . 22, 23, 46

[Don04] Stefan Donath. On Optimized Implementations of the Lattice Boltzmann Method on Contemporary High Performance Architectures. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, August 2004. Bachelor thesis. 10, 20, 33

[Igl03] K. Iglberger. Cache Optimizations for the Lattice Boltzmann Method in 3D. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, September 2003. Bachelor thesis. 6, 10, 19, 20, 28, 33

[WG00] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models*. Springer, 2000. 3

[Wil03] J. Wilke. Cache Optimizations for the Lattice Boltzmann Method in 2D. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, February 2003. http://www10.informatik.uni-erlangen.de. 6, 20, 33