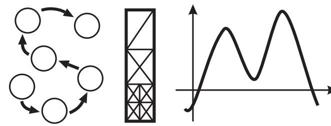
FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)



Performance Optimization Of Numerically Intensive Codes – A Case Study From Biomedical Engineering

Markus Zetlmeisl

Studienarbeit

Contents

1 Introduction **11**

1.1 Physiological Background of Seizures 11
1.2 Background of the Project 14
1.3 Structure of the Thesis 14

2 Simulation of Bioelectric Fields **15**

2.1 Forward and Inverse Problem 15
2.2 Solving the Inverse Problem 15
2.3 Solving the Forward Problem 16
2.3.1 Basic Physical Equations 16
2.3.2 Finite Difference Method (FDM) 17
2.3.3 The Reciprocity Theorem 19
2.3.4 Finite-Difference Reciprocity Method (FDRM) 20

3 The SOR Method **23**

4 Modern Microprocessors **25**

4.1 RISC and CISC 25
4.2 Processor Operation Basics 25
4.3 Pipelines 26
4.4 Branch Prediction 27
4.5 Load/store Architecture 27
4.6 Outstanding Loads 28
4.7 Superscalarity 28
4.8 Out-of-order Execution 28
4.9 Caches 29
4.9.1 Improving Memory Performance 29
4.9.2 Cache Basics 30
4.9.3 Cache Organization 30

5 Architectures and Tools **33**

5.1 Architectures 33
5.2 Compilers 33
5.3 Performance Analysis Tools 34

6 The Original Code **37**
6.1 SOR in the Code . . . 37
6.2 Red/black Iteration . . . 37
6.3 Data Structures . . . 38
6.4 Description of the Code . . . 38
6.5 Adaption of the Code . . . 38

7 Code Optimization **43**
7.1 Limiting the Processing to Head Points . . . 43
7.1.1 Precalculation of the Central Coefficient . . . 43
7.1.2 Dynamic Loop Boundaries . . . 44
7.1.3 Results . . . 45
7.2 Linear versus Red/Black Iteration . . . 49
7.3 The Influence of Casts . . . 50
7.3.1 Right-Hand Side Casts . . . 50
7.3.2 Standardizing Data Types . . . 50
7.4 The Inner `If`-command . . . 51
7.4.1 Costs of the Inner `If` . . . 51
7.4.2 Avoiding the Inner `If` . . . 52
7.5 Influence of the RHS Check . . . 55
7.6 Static Array Sizes . . . 55
7.7 Different Data Arrangements . . . 57
7.8 Padding . . . 58
7.9 Changes in the Inner Loop . . . 58
7.9.1 Using Additional Pointers . . . 58
7.9.2 Moving the RHS Part . . . 59
7.9.3 Moving the Central `If` . . . 59
7.10 1D Representation of the Problem . . . 60
7.10.1 Basic Idea . . . 60
7.10.2 Checking for Margin Points . . . 61
7.10.3 Improving a Linear Iteration . . . 61
7.10.4 Applying 1D Representation to Red/Black . . . 63
7.11 Using Indices for Coefficients . . . 64
7.12 Fine Tuning . . . 65

8 Blocking **69**
8.1 Basic Idea of Blocking . . . 69
8.2 Results . . . 70

9 Summary **73**
9.1 Overview of the Optimizations . . . 73
9.2 Most Successful Steps . . . 74
9.3 Overall Gain . . . 74

A Computer Architectures **79**
A.1 Pentium Pro 79
A.2 Alpha 21164 79
A.3 Alpha 21264 80
A.4 AMD Athlon 80

B Source Code of the Final Version **81**

List of Tables

4.1 Typical access times and sizes in the memory hierarchy (from [5], table 3-1) . . 30

7.1 Specific results for cc precalculation and dynamic loop boundaries on the Pentium Pro 46
7.2 Cycles per iteration for cc precalculation and dynamic loop boundaries on different architectures 47
7.3 Number of grid points to process applying dynamic loop boundaries with different orientations 47
7.4 Preparation costs for cc precalculation and dynamic loop boundaries on the PPRO 48
7.5 Influence of RHS casts on the Pentium Pro 50
7.6 Cycles per iteration using only one floating point data type on different architectures 51
7.7 Cycles per iteration using different data arrangements 57
7.8 Cycles per iteration with different modifications of the inner loop 59
7.9 Cycles per iteration using different margin detection methods for the 1D representation 61
7.10 Cycles per iteration with different modifications of the 1D cluster-based version 62
7.11 Cache behavior for data arrangements on the Pentium Pro 63
7.12 Cycles per iteration of different versions of the 1D cluster code with modified data arrangement 63
7.13 Cycles per iteration of 3D and 1D versions of the code 64
7.14 Cycles per iteration using different indexing methods 64
7.15 Cache analysis of the index versions on the Pentium Pro 66
7.16 Cycles per iteration using some fine tuning 66

8.1 3D blocking technique on the Pentium Pro 70

9.1 Summary table for the optimization steps used in this work 73
9.2 Comparison of the original and final code version using a bigger data set . . . 74

List of Figures

1.1 Inhibitory feed-forward and feed-back neuron configuration, taken from [2] . . 13

2.1 Example grid point and its neighbors (from [9]) 18
2.2 Reciprocity theorem (figure from [9]) 20
2.3 Overview over the whole localization algorithm using FDRM 22

4.1 A typical processor pipeline 27
4.2 Out-of-order execution architecture 29

6.1 Internal array structure in the original code 39
6.2 Structure of the original code 40
6.3 Red iteration part of the original code 41

7.1 Visualization of dynamic loop boundaries (2D) 44
7.2 cc precalculation and dynamic loop boundaries on different architectures . . . 47
7.3 Inner loops and overall processing time for the cc precalculation and the dynamic loop boundaries version 48
7.4 Linear versus red/black iteration 49
7.5 Cycles per iteration ratio avoiding the RHS casts 51
7.6 Cycles per iteration ratio avoiding the inner `if` on different architectures . . . 53
7.7 Visualization of the cluster method 53
7.8 Cycles per iteration ratio of the cluster-based version on different architectures 54
7.9 Cycles per iteration ratio without the RHS check on different architectures . . 55
7.10 Cycles per iteration ratio using static data structures on different architectures 56
7.11 Different data arrangements 57
7.12 Data access pattern when updating point i using a 1D representation 60
7.13 Clusters in the 1D case 62
7.14 Red iteration part of the 1D code 65
7.15 Cycles per iteration ratio after fine tuning 67
7.16 Cycles distribution in the red iteration part of the final version 68

8.1 Visualization of a 2D blocking method 71

9.1 Cycles per iteration ratio of the final code compared to the original code . . . 75

Chapter 1

Introduction

The American Epilepsy Society defines epilepsy as “a tendency toward recurrent seizures unprovoked by systemic or neurologic insults”, where a seizure is defined as the “clinical manifestation of an abnormal and excessive excitation and hypersynchronous discharge of a population of cortical neurons” [2]. It is estimated, that new-onset seizures occur about 80 per 100,000 people per year. About 60% of these people will have epilepsy. The prevalence of this disease is 0.5-1%. Several types of seizures are distinguished according to the “International Classification of Epileptic Seizures”. On the top level of this classification it distinguishes between *partial seizures*, which initiate locally, *generalized seizures* without localized onset and *unclassified seizures.*

The objective of the project this work is part of is the localization of regions responsible for the seizure. Therefore, the partial seizures are most interesting here. This kind of seizure is subclassified into *simple partial seizures*, *complex partial seizures* and *secondary generalized partial seizures.* The difference between simple and complex is the state of consciousness of the patient.

In simple seizures, the consciousness is preserved. They can lead to different symptoms: Motor seizures e.g. result in stiffening, twitching or jerking. Sensory seizures lead to haluzinations and illusions that can affect any sensory modality. Furthermore, autonomic seizures change the autonomic activity (heart or breath rate, sweating). Psychic seizures can lead to “dream states” or spontaneous emotions. Isolated simple seizures might not be realized as seizures at all, because many symptoms can also be caused by other diseases.

Complex partial seizures lead to an impaired consciousness, cognitive function and recall and last about 15 seconds to 3 minutes. Usually automatic movements (mouth and upper extremities), vocalization (e.g. repeating a phrase) and even complex acts can be observed. After the seizure, confusion sets in for about 15 minutes.

See [2] for a detailed analysis of the disease.

1.1 Physiological Background of Seizures

This section is based on [1] and [2]. Information in organisms is mainly transmitted and processed by *neurons.* Neurons are specialized cells which are interconnected. The unit of information in neurons is the *action potential*, a depolarization of the cell in an all-or-none fashion. The number of action potentials generated by a neuron carries the information. The connections to other neurons are called *axons* that are ending in so called *synapses.* In the

synapses, neurotransmitter substances are kept in vesicles and, in case of an action potential, are poured into the intercellular gap. Receptor channels in the receiving neurons can be opened by these substances and ions flow in or out of the neuron leading to postsynaptic potentials. This can be excitatory or inhibitory, mainly depending on the neurotransmitter. This means, an incoming action potential can raise or lower the excitability of the neuron. The receiving neuron generates an action potential itself, when it is excited, i.e. a certain potential threshold is reached. So the action potential generation depends mainly on the ratio and rate of excitatory and inhibitory influences of the incoming action potentials.

The human cortex consists of six layers with two types of neurons: The *projection neurons* like pyramidal neurons are connected to neurons in distant areas of the brain. *Interneurons* like basket cells on the other hand are locally connected. The projection neurons usually have excitatory synapses, while the interneurons form inhibitory synapses on other interneurons and projection neurons. Neurons are often connected to form feed-back or feed-forward loops. In the first case, a neuron is looped back to itself (maybe with some other neurons in between) causing itself to be inhibited or excited by its own action potential. Feed-forward interconnections provide an additional connection from one neuron to another. Depending on the runtime difference, certain effects can be achieved by this configuration. Recent work suggests, that synchronization of large groups of neurons is possible by interconnections of this kind (compare [2]). See figure 1.1 for an example interconnection (inhibitory). The $+$ synapses indicate excitatory, the $-$ synapses inhibitory influence.

The neuron excitability has a lot of influences that can also be considered as a possible cause of seizures. E.g.:

- Ion channel type, number and distribution
- Biochemical modification of receptors
- Modulation gene expression
- Extracellular ion concentration
- Modulation of neurotransmitter metabolism
- Neuronal network configuration

The beginning of a seizure is characterized by high-frequency bursts of action potentials and a hypersynchronization of a neuronal population. This occurs locally in partial seizures. It can affect the surrounding neurons and spread. Usually this is inhibited by surrounding inhibitory neuron connections.

The real reasons for seizures are unknown. Several theories exists about how a neuronal network can become hyperexcitable and how the upset of balance of exhibitory and inhibitory influence in the brain can be explained (see [2]). About 50% of the patients with severe head injury suffer from a seizure disorder. But this often develops after a time without problems (several week, months or years), during which the brain seems to change.

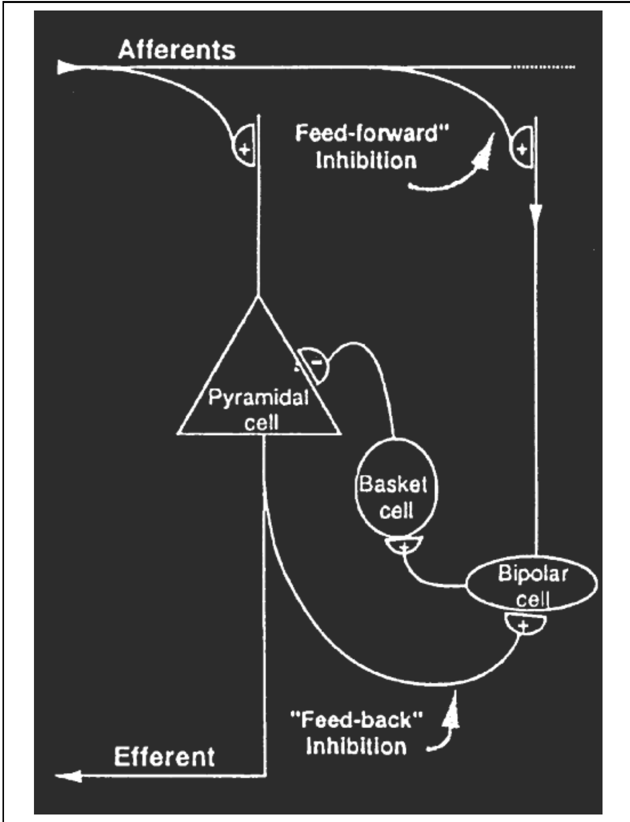


Figure 1.1: Inhibitory feed-forward and feed-back neuron configuration, taken from [2]

1.2 Background of the Project

The project this work is part of wants to localize the epileptic center within the brain of patients, where the center has to be removed by surgery. To achieve this, an *Electroencephalogram* (EEG) examination is carried out: Electrodes placed on the head scalp measure the potentials of cortical neuronal dendrites near the brain's surface over time. The goal of the project is to find the center of the disease using this potential distribution during a monitoring phase where the patient is hospitalized and a computer tomography (CT) and/or magnetic resonance (MR) record of the patient's head.

The objective of this work is to optimize the numerically most intensive code of this localization process, a 3D iteration, to find out whether the localization is feasible on mainstream computers which is important for clinical applications. The optimization is not limited to the application of this project. It can be used in several realms of bioelectric field simulation. The ideas can be applied in other fields of code optimization, mainly numerical iterations.

1.3 Structure of the Thesis

Chapter 2 introduces the theoretical background of the simulation of bioelectric fields and describes the discretization of the governing differential equation, yielding to a system of linear equations. Then, chapter 3 presents the SOR method applied in the project to solve large systems linear equations. Afterwards, chapter 4 gives an introduction to the features of modern microprocessor architectures. These have to be kept in mind for code optimization. Chapter 5 presents the computer architectures used for this work and the tools applied to do performance measurements, while chapter 6 explains the original code which was the basis of the optimization.

The next chapter (chapter 7) is the main part of this thesis and presents the code optimizations applied to the code and their results. Chapter 8 is an add-on to this and shows an additional memory access technique tested in the code. Finally, chapter 9 summarizes the successful optimization steps and presents some final performance measurements.

In the appendix, a detailed specification of the architectures used in the work is given. Furthermore, the source code of the final code version is printed.

Chapter 2

Simulation of Bioelectric Fields

This chapter gives an overview of the algorithm used in the project and its mathematical background. It is based on [9] which gives a more detailed description of these methods.

First, some basic terms of dipole localization are explained (section 2.1). Then, the basic idea of solving the inverse problem is described in more detail (section 2.2). Afterwards, section 2.3 describes the equations and solution methods to solve the forward problem which is a very important part in the whole procedure and target of the optimization done in this work. Note, that some of the methods described in this chapter work well with arbitrary current sources and sinks. But we concentrate only on dipoles as bioelectric sources. A dipole is defined by the position of its two poles and its current magnitude.

2.1 Forward and Inverse Problem

Two basic terms have to be distinguished in bioelectric field simulation:

- The forward problem denotes the procedure of calculating the potentials at the EEG electrodes given the current sources and sinks (or a dipole in our case) in the head.
- The inverse problem on the other hand is to localizes the dipole by seeking the best-matching dipole position given the potentials at several distinct spots. In our case, these are the potentials measured by the scalp electrodes in an EEG examination. So, by solving the inverse problem, it is possible to find the region in the brain responsible for seizures of the patient.

2.2 Solving the Inverse Problem

Basically, the inverse problem is solved by an optimization approach: Start with a random dipole position and orientation. Then, a forward problem is solved to obtain the potential differences at the scalp electrodes, induced by this dipole. The calculated potentials are compared to the actually measured ones. Afterwards, the position and orientation of the dipole is modified and the forward problem is solved again with the new dipole parameters. These steps are repeated until the optimum dipole parameters are found. This means, that the difference between measured and calculated clamp potentials is minimal. Typically, several hundred forward problems have to be solved to localize a dipole. Therefore, it is important to solve the forward problem efficiently.

2.3 Solving the Forward Problem

There are several methods to solve the forward problem: The BEM (*boundary element method*) uses usually a few isotropically conducting compartments, where "isotropic" denotes the fact, that a tissue shows the same conductivity in all directions. To keep the number of compartments low, tissues with different conductivities are often comprised in one compartment. But some tissues of the head are anisotropic, i.e. showing different conductivities for different directions. This cannot be modeled with this method.

Other methods like FEM (*finite element method*) and FDM (*finite difference method*) are able to model anisotropy and a larger amount of different tissues. FDM is used in this project. There, the head is discretized into small voxels (cubes) by utilizing a cubic grid. Each voxel is regarded to be isotropic and can be assigned a certain conductivity. The need for reasonable resolutions leads to a high number of grid points, which eventually means a large system of linear equations.

Solving this set of linear equations for each dipole position in the inverse problem would be too time consuming and not be accepted for a clinical application. Therefore the FDRM (*finite difference reciprocity method*) is applied in this project. It uses FDM and the *reciprocity theorem* to limit the number of systems of linear equations for the whole localization procedure to $m(m-1)$, where m denotes the number of EEG electrodes used.

2.3.1 Basic Physical Equations

The basic formula for the simulation of bioelectrical effects is the Poisson differential equation:

$$\mathrm{div}(\bar{\bar{\sigma}}(\bar{r})\mathrm{grad}(V(\bar{r}))) = -I(\bar{r}) \tag{2.1}$$

$V(\bar{r})$ denotes the potential at a point $\bar{r}$. $\bar{\bar{\sigma}}(\bar{r})$ is the conductivity tensor at point $\bar{r}$. $I(\bar{r})$ is the current at point $\bar{r}$. In our case, the right hand side is

$$-I\delta(\bar{r}-\bar{r_2}) + I\delta(\bar{r}-\bar{r_1})$$

where δ denotes the 3D delta function and $\bar{r_1}$ resp. $\bar{r_2}$ identify the points of the current source resp. sink. I is the magnitude of the dipole's current.

In order to solve this equation numerically, the head model is discretized into isotropic compartments. These are equally sized cubes in this project. A certain constant conductivity is assigned to every compartment. So the conductivity tensor becomes a scalar σ. We can set up equation (2.1) for every compartment:

$$\frac{\partial}{\partial x}(\sigma\frac{\partial V}{\partial x})+\frac{\partial}{\partial y}(\sigma\frac{\partial V}{\partial y})+\frac{\partial}{\partial z}(\sigma\frac{\partial V}{\partial z}) = -I\delta(x-x_2)\delta(y-y_2)\delta(z-z_2)+I\delta(x-x_1)\delta(y-y_1)\delta(z-z_1) \tag{2.2}$$

Furthermore, we need another basic formula, describing the situation at the interface of two compartments (elements of the grid). This is called the *Neumann boundary condition* (2.3) and formalizes the fact, that the current leaving a compartment in a certain direction has to be equal to the current entering the corresponding neighbor compartment. This equation is set up at the interface point $\bar{r_0}$ of two compartments, where n is the normal vector on the surface.

$$\lim_{\bar{r}\to\bar{r_0}|\bar{r}\in A} \sigma_A \frac{\partial V}{\partial \bar{n}} = \lim_{\bar{r}\to\bar{r_0}|\bar{r}\in B} \sigma_B \frac{\partial V}{\partial \bar{n}} \tag{2.3}$$

A and B are compartments with conductivities σ_A and σ_B, respectively. This Neumann condition also defines a state for the scalp-air boundary. As no current should leave the head, we can write:

$$\lim_{\bar{r} \to \bar{r_0} | \bar{r} \in scalp} \sigma_{scalp} \frac{\partial V}{\partial \bar{n}} = 0 \tag{2.4}$$

2.3.2 Finite Difference Method (FDM)

The differential equation (2.1) is discretized using a finite difference approximation. The method used for this transformation is called *box integration scheme*. Just imagine the head model divided into equally sized isotropic cubes by using a cubical grid. The potential at every cube's center is regarded. The internode distance should be h. Look at figure 2.1 and let us consider an example node P with the coordinates $(0, 0, 0)$ and its neighbors in the grid $Q_i (i = 1...6)$. Furthermore, $N_i (i = 1...6)$ are the midpoints of PQ_i. These points are located on the surfaces of the cubes P and Q_i and are the interface points to the cube's neighbors. So the Neumann condition has to be fulfilled at each of these points.

Notice that the conductivity within a cube is constant. So a cube is an isotropic compartment and (2.2) can be set up for every cube. Now we integrate (2.2) over the cube. The first summand becomes:

$$\int \int \int_{cube} \frac{\partial}{\partial x} (\sigma \frac{\partial V}{\partial x}) dx\, dy\, dz =$$
$$\lim_{x \to \frac{h}{2}} \int \int \sigma \frac{\partial V}{\partial x} dy\, dz - \lim_{x \to -\frac{h}{2}} \int \int \sigma \frac{\partial V}{\partial x} dy\, dz$$

Using the midpoint rule, the integrals can be approximated and the result is:

$$\lim_{x \to \frac{h}{2}} h^2 \sigma \frac{\partial V}{\partial x} - \lim_{x \to -\frac{h}{2}} h^2 \sigma \frac{\partial V}{\partial x} \tag{2.5}$$

Refer to figure 2.1 again. Let us denote the potential at our currently regarded node P as V_P and its conductivity σ_0. The potential at the interface point to the right neighbor cube in x-direction (called Q_1) is V_{N_1}. The conductivity of this neighbor is σ_1. Now, we can set up the boundary condition at the interface point N_1 in the following way:

$$\lim_{x \to \frac{h}{2}} \sigma_0 \frac{\partial V}{\partial x} = \lim_{x \to \frac{h}{2}} \sigma_1 \frac{\partial V}{\partial x}$$

The derivative can be written as a difference:

$$\sigma_0 \frac{V_{N_1} - V_P}{\frac{h}{2}} = \sigma_1 \frac{V_{Q_1} - V_{N_1}}{\frac{h}{2}} \tag{2.6}$$

Solving this equation for V_{N_1} leads to:

$$V_{N_1} = \frac{\sigma_0 V_P + \sigma_1 V_{Q_1}}{\sigma_0 + \sigma_1} \tag{2.7}$$

We can replace V_{N_1} in the left side of (2.6) with (2.7). So the left term can be written as

$$\lim_{x \to \frac{h}{2}} \sigma_0 \frac{\partial V}{\partial x} = \sigma_0 \frac{V_{N_1} - V_P}{\frac{h}{2}} = \frac{2}{h} \frac{\sigma_0 \sigma_1 (V_{Q_1} - V_P)}{\sigma_0 + \sigma_1} \tag{2.8}$$

Figure 2.1: Example grid point and its neighbors (from [9])

We can substitute $\lim_{x\to\frac{h}{2}} \sigma_0 \frac{dV}{dx}$ of the first part of (2.5):

$$\lim_{x\to\frac{h}{2}} h^2 \sigma_0 \frac{\partial V}{\partial x} = 2h \frac{\sigma_0 \sigma_1}{\sigma_0 + \sigma_1}(V_{Q_1} - V_P).$$

Repeating these steps with the left neighbor of P in x-direction (Q_2) substitutes the second part of (2.5).

Repeating the whole procedure for the integration in y- and z-direction, we finally obtain a discretized approximation of the Poisson differential equation (2.2) for every cube in the grid:

$$\sum_{i=1}^{6} \alpha_i V_{Q_i} - \alpha_0 V_P = I_P \tag{2.9}$$

with

$$\alpha_i = 2h \frac{\sigma_0 \sigma_i}{\sigma_0 + \sigma_i} \tag{2.10}$$

and

$$\alpha_0 = \sum_{i=1}^{6} \alpha_i \tag{2.11}$$

The right hand side is

$$I_P = \int\int\int_{cube} (-I\delta(x - x_2)\delta(y - y_2)\delta(z - z_2) + I\delta(x - x_1)\delta(y - y_1)\delta(z - z_1))dx\,dy\,dz$$

This simply means that I_P is I or $-I$ for cubes with a current monopole and 0 for all other cubes.

Now we have one equation of type (2.9) for every node of the head model. As this equation is linear, we have a system of linear equations that describes the forward problem. This means, we are now able to calculate the potential for every node of the discretized model given the conductivities σ of every node in the grid and the location of a dipole of current I.

2.3.3 The Reciprocity Theorem

In order to find the dipole position matching the measured EEG potentials, we have to run an optimization algorithm. We have to determine the resulting potentials at the electrodes and compare these to the real values for every optimization step of the inverse problem. This means, we would have to solve a large system of linear equations of the kind described in the section before. Even with the power of modern computers this is not feasible in a reasonable amount of time with a sufficient resolution. The reciprocity theorem helps to simplify the solution of the forward problem (i.e. to obtain the expected potentials at the electrode clamps).

Have a look at figure 2.2(a). The dipole is represented by a current source and sink with a distance of $2h$. You see two electrodes A and B measuring the potential difference V_{AB} induced by a current dipole located in x-direction in the head. This represents our real application.

In figure 2.2(b), we see the same configuration, but we now have a current I_{AB} between the electrodes and measure the potential V_{r_x} at the dipole.

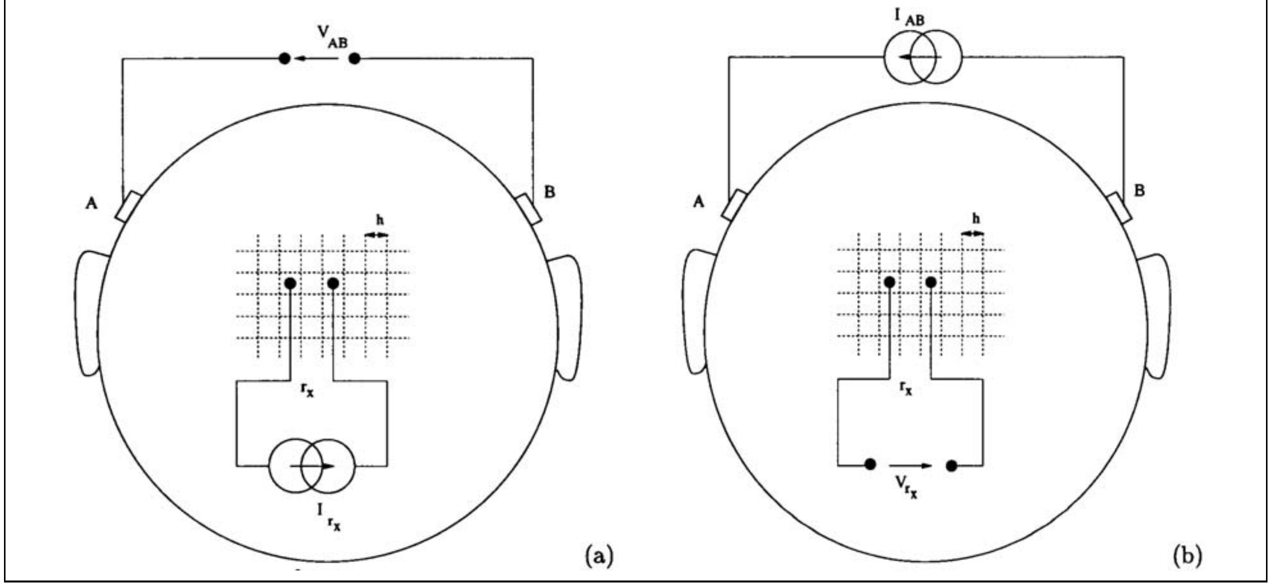


Figure 2.2: Reciprocity theorem (figure from [9])

The theorem combines these two cases:

$$V_{AB}(I_{r_x})I_{AB} = V_{r_x}(I_{AB})I_{r_x}$$

or

$$V_{AB}(I_{r_x}) = \frac{V_{r_x}(I_{AB})I_{r_x}}{I_{AB}} \tag{2.12}$$

This fact leads to a simple way to solve the forward problem.

2.3.4 Finite-Difference Reciprocity Method (FDRM)

Assume, we calculate the potential distribution generated by a current between the electrodes AB in figure 2.2(b) and store it in a file. We can easily obtain the potential $V_{r_x}(I_{AB})$ in x-direction at position i, j, k:

$$V_{r_x}(I_{AB}) = V_{i+1,j,k}(I_{AB}) - V_{i-1,j,k}(I_{AB})$$

The two values on the right hand side can be found in the file created before. We can use the same method to calculate $V_{r_y}(I_{AB})$ and $V_{r_z}(I_{AB})$. Set $I_{AB} = 1A$ and $2hI_{r_x} = 2hI_{r_y} = 2hI_{r_z} = 1Am$. Now we can use (2.12) to get the potential difference at AB caused by a dipole at i, j, k oriented along the x-, y- or z-coordinate:

$$V_{AB}(I_{r_x}) = \frac{V_{i+1,j,k}(I_{AB}) - V_{i-1,j,k}(I_{AB})}{2h}$$

$$V_{AB}(I_{r_y}) = \frac{V_{i,j+1,k}(I_{AB}) - V_{i,j-1,k}(I_{AB})}{2h}$$

$$V_{AB}(I_{r_z}) = \frac{V_{i,j,k+1}(I_{AB}) - V_{i,j,k-1}(I_{AB})}{2h}$$

For a dipole with orientation $\bar{d} = (d_x, d_y, d_z)^T$ we can write:

$$V_{AB}(\bar{r}, \bar{d}) = \bar{L}'(\bar{r}) \cdot \bar{d}$$

with $\bar{L}'(\bar{r}) = (V_{AB}(I_{r_x}), V_{AB}(I_{r_y}), V_{AB}(I_{r_z})) \in \mathbf{R}^{3\times 1}$.

For m electrodes there is a maximum of $m-1$ electrode pairs for which the potential differences are independent. Therefore we can repeat this process for $m-1$ electrode pairs. So we can simplify the dipole localization algorithm by calculating $m-1$ potential distributions given by a current between the corresponding electrodes, i.e. solving $m-1$ systems of linear equations and saving the results. These distributions only depend on the head geometry of the patient and the position of the electrodes. Therefore, this can be done before the optimization process. We can use these distribution files and the method described above during the optimization to calculate the potentials at the electrodes by mainly a simple matrix-vector multiplication (for details see [9]). Solving the forward problem with the help of the reciprocity theorem is called the “Finite-Difference Reciprocity Method (FDRM)”.

This procedure has another advantage: The distribution data in the file(s) can be reused for several source localization attempts of the same person. This is important in the clinical application, because it is necessary to verify the dipole location by repeating the process with different electrode data sets (different seizures).

Note that FDRM only provides the potentials at the predefined electrode positions and not of the entire head model.

Now, we can avoid the numerically intensive system of linear equation in the optimization phase of the dipole localization. But still, solving the system $m-1$ times during the preparation step is a time consuming task in the whole process and an efficient implementation for it is necessary.

See figure 2.3 to see an schematic overview of the whole process of dipole localization using FDRM.

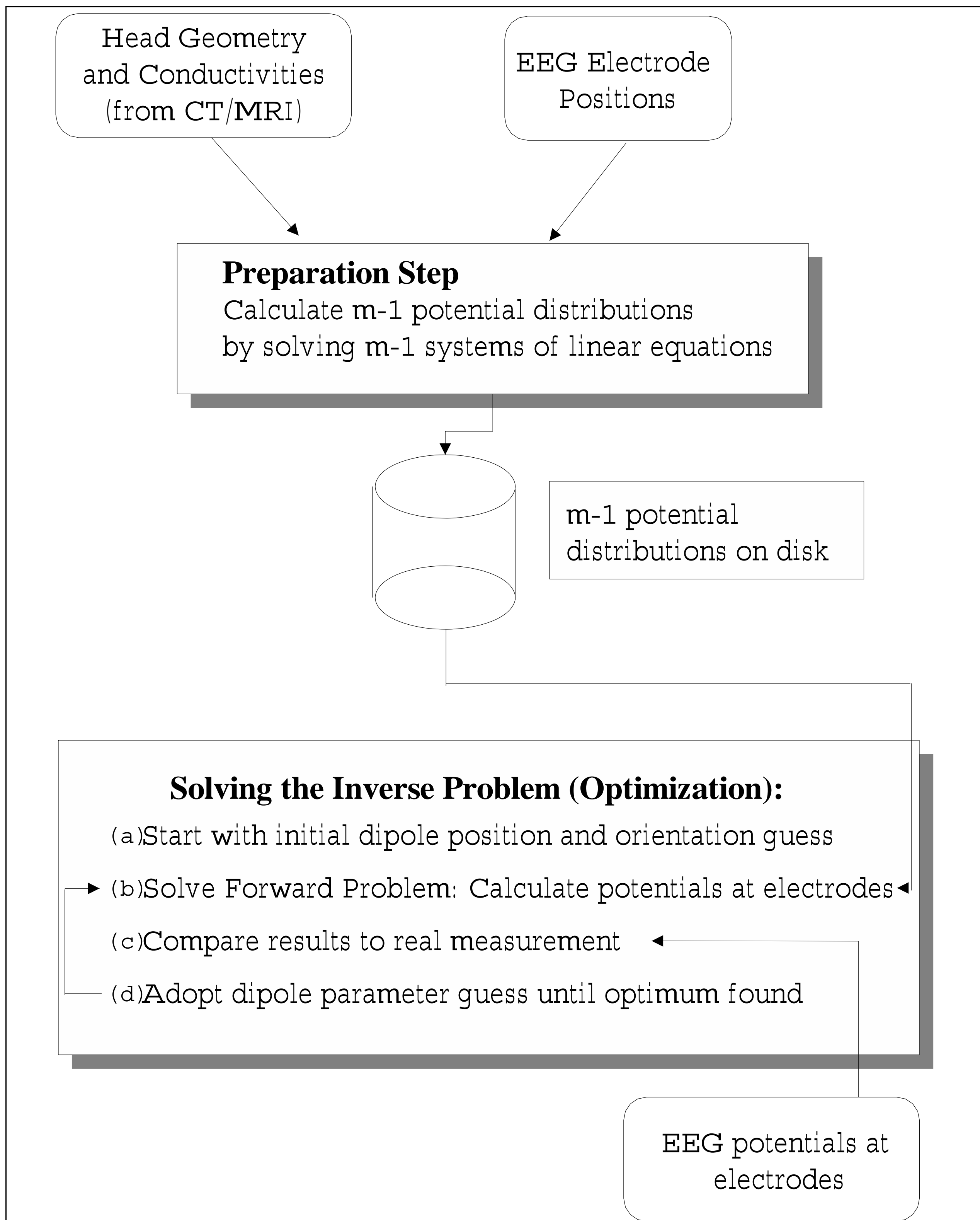


Figure 2.3: Overview over the whole localization algorithm using FDRM

Chapter 3

The SOR Method

The system of linear equations for FDRM is solved by an iterative solution method called SOR (*successive over-relaxation*). This is a well-known method for solving large systems of linear equations that cannot be solved directly on the computer. Setting up the complete matrix for the system with a resolution of 63 nodes per dimension (this resolution was used in the project), leads to 250046 unknowns and a matrix size of 250046^2 elements. But looking at equation (2.9), we see that there are only up to seven non-zero entries in a row of the matrix. So the matrix could be stored in a compressed way. But using a direct solver would gradually fill in matrix elements that are zero in the beginning, blowing up the memory usage of the solver. This chapter describes the idea behind SOR and some related methods for linear systems. Refer to [3] more information.

The goal of all methods is to solve a system of linear equations

$$Ax + b = 0, \qquad A \in \mathbf{R}^{n \times n}, x, b \in \mathbf{R}^n \tag{3.1}$$

The matrix A must be regular and x is a vector of n unknown values.

This system is a matrix-vector representation of n linear equations:

$$\sum_{j=1}^{n} a_{ij} x_j + b_i = 0, \qquad (i = 1, 2, ...n)$$

Furthermore we assume that all diagonal elements of the matrix A are not zero. This can be fulfilled be reordering the equations. So we can solve the i-th equation for x_i:

$$x_i = -\frac{1}{a_{ii}} \left[\sum_{j=1, j \neq i}^{n} a_{ij} x_j + b_i \right], \qquad (i = 1, 2, ...n)$$

This system of equations defines a linear transformation from $\mathbf{R}^n$ to $\mathbf{R}^n$. And the solution x of the original linear equation (3.1) is a fix point of it. So we can derive an iteration method from this fact:

$$x_i^{(k+1)} = -\frac{1}{a_{ii}} \left[\sum_{j=1, j \neq i}^{n} a_{ij} x_j^{(k)} + b_i \right], \qquad (i = 1, 2, ...n; k = 0, 1, 2, ...)$$

One method called *Jacobi's method* is to calculate the new values of x by only using values of the iteration step before like in the above formula. From a programmer's view, we can see

it as to allocate two blocks of memory $b1$ and $b2$ for the result vector. We start with initial values for x in $b1$ and calculate the new values for x by reading the values of $b1$ and storing the results in $b2$. When a complete set of values is calculated, we just change the role of the memory blocks and do another iteration.

You can also think about working in one memory block only and overwrite a memory position as soon as the corresponding value is updated. So the iteration step $k+1$ is done by using partly old values (values of iteration k) and new values (iteration $k+1$). This is know as the *method of Gauss-Seidel*. It can be written formally:

$$\tilde{x}_i^{(k+1)} = -\frac{1}{a_{ii}} \left[\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} + b_i \right],$$

$$(i = 1, 2, ...n; k = 0, 1, 2, ...)$$

and set

$$x_i^{(k+1)} = \tilde{x}_i^{(k+1)}$$

The correction of x_i is

$$\Delta x_i^{(k+1)} = \tilde{x}_i^{(k+1)} - x_i^{(k)}$$

Practical analysis shows that it can be useful to multiply the correction of every x_i with a constant factor, the relaxation factor, and update x_i by adding this new correction to the old value. This method often converges much faster for suitable ω. So updating the value of x_i in the following way

$$x_i^{(k+1)} = x_i^{(k)} + \omega \cdot \Delta x_i^{(k+1)}$$

with the relaxation factor ω is called the SOR-method. The whole SOR-iteration can be summarized in the equation

$$x_i^{(k+1)} = (1-\omega) x_i^{(k)} - \frac{\omega}{a_{ii}} \left[\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} + b_i \right], \tag{3.2}$$

$$(i = 1, 2, ...n; k = 0, 1, 2, ...)$$

There are several principles about choosing the best relaxation factor, but in this project, the factor is determined by hand. Furthermore, there is some theory in [4] and [3] to make sure that the iteration converges at all. This theory is not covered by this treatise, because the SOR-method works fine with the manually determined factor in this project.

Chapter 4

Modern Microprocessors

To optimize code that solves numerical problems, it is necessary to have a basic understanding of how modern microprocessors work internally. Therefore this chapter gives an introduction to the features built in contemporary processors to achieve performance. For a detailed coverage of architectural features important for code optimization see [7], [5], [6] and [10].

4.1 RISC and CISC

You usually distinguish between RISC (reduced instruction set computer) and CISC (complex instruction set computer) processors. CISC processors, on the one hand, have a lot of (several hundreds of) high level machine instructions with a lot of addressing modes for the operands and varying instruction length and processing times. RISC processors, on the other hand, usually have significantly less than a hundred very simple instructions that are equal in length, so that an operation that needs only one instruction on a CISC machine might take a few on a RISC architecture. But the few instructions of a RISC are implemented very efficiently in hardware. This is possible due to their limited number and the equal instruction length. The RISC architecture proved to be more efficient for numerically intensive codes and is found in every high performance computer nowadays. The items mentioned in this chapter describe features of RISC processors. The two CISC processors used in this thesis break their complex machine instructions into simpler RISC style instructions internally. Thus, the features described here are valid for all processors involved in this work.

4.2 Processor Operation Basics

Processors still behave in the way von Neumann proposed. They consist of a memory unit that manages the access to the main memory which stores data and instructions of the programs currently running. Furthermore, they have a instruction decode unit that determines the current command and an execution unit that carries out the actual arithmetic or logical operation. Usually, the instructions are executed in the same order in which they occur in the main memory. But there are conditional and unconditional jumps that break this sequential order. The processing of a machine code instruction can be divided into the following stages:

1. Instruction fetch

   The processor fetches the next instruction from the memory system.

2. Instruction decode

   The processor determines the kind of the instruction.

3. Operand fetch

   The operands of the instruction are fetched from the registers or memory system.

4. Execute

   The instruction is carried out.

5. Write-back

   The result of the execution is written back to a register or to the memory system.

Depending on the processor, there might be a different number of stages. This should be regarded as an example. Ideally, every stage takes one cycle, so that the whole operation is finished after five cycles in the given example.

4.3 Pipelines

These stages of an instruction depend on each other and therefore cannot be executed in parallel. However, it is possible to perform different stages of different instructions at the same time. So, for example, when an instruction 1 is in stage two, we can already process stage one of the next instruction 2 and so on. This leads to the idea of a pipeline execution that is implemented in almost every modern processor. Refer to figure 4.1 to see the time schedule of a typical pipeline. So in an ideal operation, every stage of the instruction takes one cycle and a new instruction can be started in every cycle. This also means, that in every cycle one instruction is finished and, at a given clock cycle, five different instructions are in the pipeline at different stages and are processed in parallel. So the processor runs five times faster than without a pipeline.

In reality, there are some occurrences that block the pipeline:

- Some stages of the pipeline might take longer than others. It might be possible that the operand fetch takes a long time due to cache misses (see section 4.9). Or it might take a long time to compute the operand's address.

- There might also be some dependency between the instructions in the pipeline, e.g. one instruction might need the result of an instruction that is still being processed in the pipeline.

- Jumps also disturb the smooth run of a pipeline, because the next instruction address depends on the result of the jump condition. It is possible to wait until the condition is evaluated before allowing further instructions to enter the pipeline. This may lead to a bad performance and therefore, the pipeline is usually filled with the next sequential instructions. If the branch is to be taken, all the intermediate results of these instructions are discarded and the pipeline is restarted with the instructions of the new address. This is called a pipeline flush. It is obvious that jumps are crucial for efficient computing and therefore a lot of effort was put in avoiding these flushes (see section 4.4).

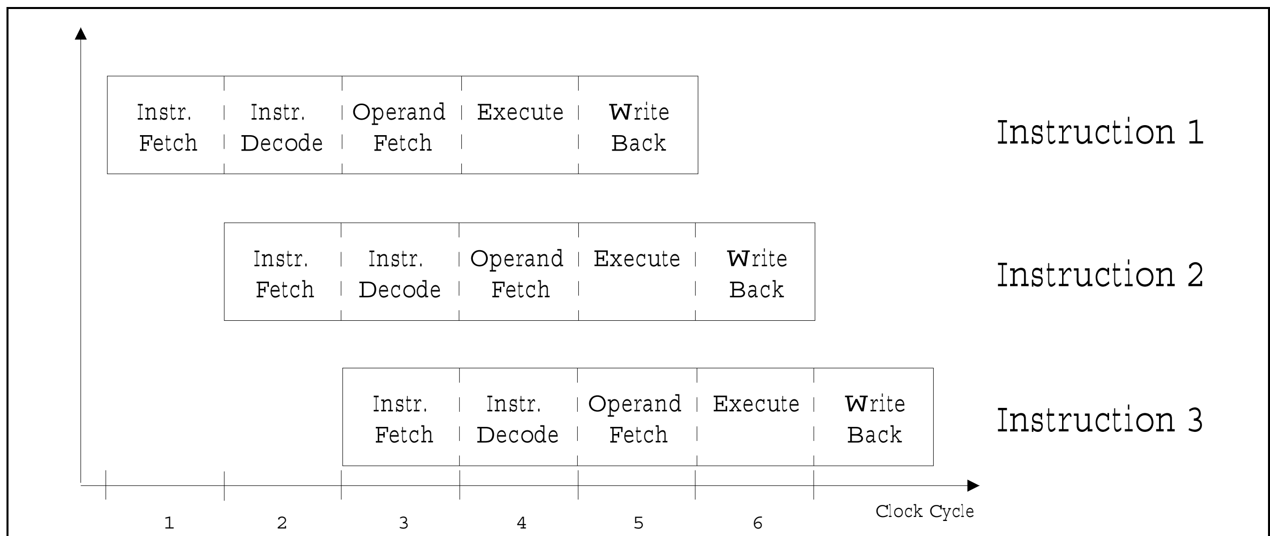


Figure 4.1: A typical processor pipeline

A lot of processors support pipelined floating point operations, meaning that this kind of operation has a different pipeline which operates in parallel to the instruction pipeline. This is due to the longer execution time of floating point calculations.

4.4 Branch Prediction

Keeping the pipeline filled is very important to achieve good performance results. Therefore a lot of effort was done by the processor designers to avoid pipeline stalls. Processors use a separate unit to predict the results of branches by observing branches at runtime and storing their behavior. So when the decode unit detects a branch instruction, it asks the prediction unit which way the branch will probably go and fills the pipeline with the corresponding instructions the unit predicts. If the prediction is wrong, the pipeline has to be flushed. A lot of sophisticated techniques have been developed for this prediction unit. The Alpha 21264, for example, can even use the behavior of the preceding branches, if it correlates with the current one (global history). It can also detect certain patterns the branch follows (see [12] for details).

4.5 Load/store Architecture

CISC instructions usually can operate on the main memory directly. This means, they can include memory addresses for operands or the results. In RISC instruction sets, the arithmetic and logic operations are performed on the fast internal processor registers only. The operands are transferred by separate load instructions to these registers and results are saved to the memory by explicit store instructions. This allows the compiler to optimize the code by deciding when to remove the data from the register, depending on the frequency the data is needed during the next operations. If the processor supports outstanding loads (see section 4.6), the compiler can decide to load a data a few cycles before it is needed. So the memory unit can fetch the data in parallel, while other operations are executed by the processor.

RISC processors usually have more than 50 registers accessible from the outside. The CISC processors used in this thesis have only a limited number of registers directly accessible by the compiler, but use more of them internally (see renaming registers in section 4.8).

4.6 Outstanding Loads

Memory bandwidth and access times are a big problem in contemporary computers. It usually takes much more than a cycle to get data from the main memory to the register (see section 4.9 for a possibility to limit this problem). This slows down the overall performance of the processor and decreases efficiency dramatically.

Modern processors allow load instructions that do not block the processor until a piece of data is in the register. It can go on processing instructions independent of the loaded data and perform the memory access in parallel. So the compiler can prefetch data several cycles before it is needed. A lot of systems even allow several of these outstanding loads at a time.

4.7 Superscalarity

To achieve more computing power, modern processors are often fitted with more than one pipeline. They have more than one decode unit, so that they can issue more than one instruction at a time. They have several floating point units to perform more than one operation in parallel and several integer units for integer and logical arithmetic. The challenge with these processors is to keep all these units busy. E.g., it is not possible to perform two arithmetic operations at a time, if one needs the result of the other. Furthermore different kinds of operations might take different numbers of clock cycles and the hardware has to guarantee that the results appear in the correct order. In processors that do not have a post-RISC pipeline (see section 4.8), it is mainly the task of the compiler to put the instructions in the best order to utilize the different pipelines most efficiently. This technique is called *software pipelining*.

4.8 Out-of-order Execution

The fact of having several pipelines leads to the difficulty of ordering the instructions in a good way to keep all the units busy. This is a very difficult part, above all, because sometimes information about the dependencies of instructions is only available at runtime (e.g. if the address of an operand is computed in the program). This problem is solved in processors of the so-called *post-RISC architecture*. They have a special buffer called *instruction reorder buffer* (see figure 4.2) capable of holding about 60 instructions. After the fetch and decode step, these processors put the instructions into this buffer. They determine the order of the execution of the instructions in the buffer on their own. So they can keep their pipelines busy. This concept is quite challenging, because the processor has to keep track of the dependencies of the instructions' operands and the registers. If there is a branch prediction, the instructions following the branch have to be marked in order to find and cancel them, if the prediction was wrong. The results are typically written into so-called *renaming registers* that are hidden. So these processors have more registers internally than they provide to the outside. The set of registers seen from the outside is a dynamically changing view of the internal registers.

The last step of the post-RISC pipeline is called *retiring*. This step has to show the results in the correct order by making the contents of the corresponding renaming registers visible and raise exceptions that might have occurred during execution. There are a lot of techniques to implement the out-of-order execution like *Scoreboarding* and the *Tomasulo's Algorithm* to manage the dependencies of the operands. [10] gives a good summary on these techniques.

The out-of-order execution is a very powerful feature. Processors with this feature also support outstanding loads (see section 4.6): Different memory accesses might take a different number of cycles to retrieve the data, due to cache effects (see section 4.9). But as soon as a data item is available the instructions waiting for it might execute, even if the the original execution order is different. So the instruction is not delayed due to a slower memory access of an independent instruction before.

4.9 Caches

4.9.1 Improving Memory Performance

Processors have become faster and faster in the last years deploying more and more internal parallelism with pipelines. Furthermore the clock rate could be raised. But another important task is to get data from and to the processor. The memory performance has not increased so much. To give a number: A memory access on today's computers takes about 100 clock cycles of the processor. The bigger the working set of an application, the more time is wasted waiting for the memory system. There are several approaches to this problem:

- Use a fast memory technology (e.g. SRAM).
- Widen the memory system, so that several bytes can be transferred at a time.
- Divide memory into different banks so that a sequential access uses all banks in a round robin pattern.
- Use a memory hierarchy (caches). This means: Use faster memory for data that is used more often.

All those items increase the price of the computer system and therefore, a reasonable mixture of these ideas is used in current computers.

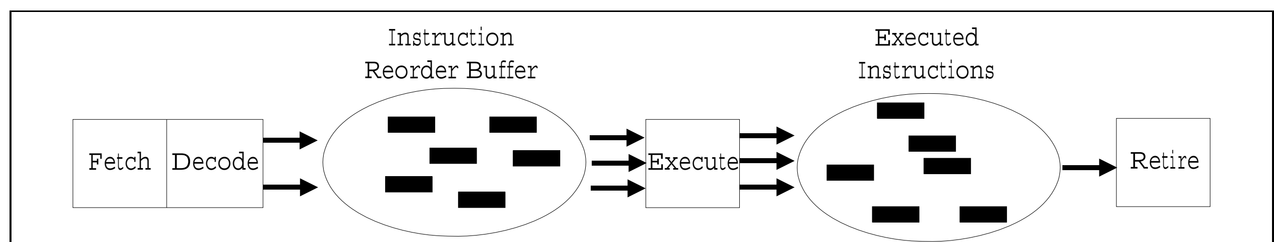


Figure 4.2: Out-of-order execution architecture

4.9.2 Cache Basics

Caches are a very common principle applied in computers to increase memory performance. Caches are fast memories put in between the processor and the main memory. The idea is to store a copy of data that is often used (locality in time). Furthermore, it should store data in the vicinity of data already accessed, because this might be accessed next (locality in space), e.g. to speed up code that works sequentially through an array.

The cache is much smaller than the main memory due to technical and economic reasons. So there must be a certain logic that decides what data resides in the cache (cache organization).

There is often more than one cache level between the processor and the main memory. Up to three cache levels (called L1, L2 and L3, where L1 is nearest to the processor) are common. The nearer to the processor, the faster and smaller is the cache memory. Together with the processor's internal registers, that can be accessed with clock speed, and the main memory, this arrangement is called memory hierarchy. (If the computer supports swapping, the swap device is also part of this hierarchy). Table 4.1 gives a short overview about some figures of a typical system.

Caches are transparent to the software. This means, that a program still uses main memory addresses to load and store data. Whether the data is in the cache or not, is unknown to the program. The hardware cares for the correct behavior. This also means, that there is no way to tell the system explicitly which data should be cached or not. This is all decided at runtime by the memory system. When the software issues a load instruction, the cache logic first looks into the cache, whether the data can be found there. If this is the case (*cache hit*), the data is returned to the processor. If not, the request is send to next level of the memory hierarchy. This is a called a *cache miss* in the current level. On a cache miss, the new data is copied to the corresponding cache level for further accesses.

But why is it important to understand the cache technique, if you do not have any explicit influence on it? It is true, that you do not have machine instructions to configure the cache, but writing software that is aware of the cache levels can improve the performance of the code dramatically. Being aware of the cache means to lay out the data structures of the program in a way that uses the cache most efficiently. Furthermore you can influence the performance by adapting the access pattern to the data with the cache in mind, if the algorithm used allows this.

4.9.3 Cache Organization

The unit of storage of a cache is the *cache line*. These lines are several bytes long and can hold coherent blocks of the main memory. Only whole lines can be replaced in the cache. This

	Access time	Size
Register	2ns	<100 words
L1 On-Chip	4ns	Tens of kilobytes
L2 Off-Chip	5ns	Hundreds of kilobytes
Memory	220 ns	Megabytes

Table 4.1: Typical access times and sizes in the memory hierarchy (from [5], table 3-1)

concept speeds up code with spacial locality: If the first byte of a data block is accessed by the program and the block is copied into the cache line, the following data items are already cached, because they are part of the same cache line and thus can be accessed faster. So caches speed up operations that go sequentially through the data.

Associativity

Caches are much smaller in capacity than the main memory. There are several methods to decide which memory block is stored in which cache line.

- Fully associative caches

  Every memory block can be stored in every cache line. This is quite complicated to implement, because the whole cache has to be searched to find a piece of data. This can be done efficiently with very small caches only. Furthermore, you have to decide which cache line has to be replaced, if the cache is full and new data has to be copied into it. So you have to implement e.g. something like a least-recently-used strategy for the cache lines.

- Direct mapped caches

  In a direct mapped cache, a hash function is applied to the memory address to find out the cache line for it (e.g. simply take all except the least significant bits of the memory address). So there is a distinct cache address for each memory location. Of course, several memory locations are mapped to the same cache line. A big advantage of this method is the easy way to check, whether a data item is in the cache: Simply apply the hash function to the address and check the corresponding cache line. But, if a piece of code uses continously two data structures at once that are, by hazard, mapped to the same line, this cache line is always replaced (*thrashing*). This leads to a very poor cache performance and has to be avoided by the programmer or the compiler.

- Set-associative caches

  This design combines the advantages of the two other designs. In a set-associative cache a hash function is applied to the memory address, too. But the position calculated by this function is not a single line but a set of typically two to four cache lines. To check, whether a data item is in the cache, the hash function is used to find the set. Then, all lines in the set are checked simultaneously. If a line has to be replaced, a kind of least-recently-used strategy is applied to the lines in a set. The big advantage of this concept is the fact, that you can use up to set-size data structures simultaneously in numerically intensive code without worrying about poor cache behavior due to bad data alignment and thrashing effects.

Behavior on Write

There are different concepts of how a cache architecture behaves on write accesses. If the data written is not in the cache, some caches first transfer the data to the cache and write it there. Some write the data directly to the main memory in this case.

Some caches do modifications of cached data only in the cache (*write back*). If a cache line has to be replaced, the line has to be saved back to memory in this case. A “modified” bit

is used to mark cache lines that have been written to. This all takes additional time, when a cache miss occurs.

Some caches do changes in the cache and transfer the store instruction further down in the hierarchy (*write through*). When the line has to be replaced, it can be simply overwritten in this case. In this case, a cache miss is not so expensive.

Physical and Virtual Caches

Caches can also be distinguished by the addresses they use. Physical caches use physical memory addresses and are placed between the MMU (Memory Management Unit) and the memory. Virtual caches use virtual memory addresses and are placed between the processor and the MMU. They allow faster access, because they do not have to wait for the memory address translation, if the data is found in the cache. But they have either to be flushed completely on a process switch or have to store a process ID for each line. Physical caches on the other hand do not care for processes. Virtual caches are better for the programmer because the data layout can be directly related to the cache. With a physical cache, you have to keep in mind that the data layout is always transformed by the MMU. So the memory pages containing a large data block of the program (larger than the memory page size of the system) might not be coherent in the physical memory/cache (at least after swapping).

Chapter 5

Architectures and Tools

This chapter describes the framework of the optimization for this project.

5.1 Architectures

The main objective of the work is to optimize a code that uses SOR to solve a specially structured system of linear equations. Real hardcore optimization is only possible, when knowing the detailed architecture of a computer system and adopting the code accordingly. But the task of this project is to find optimizations that improve the performance of the code on a bunch of contemporary architectures. The reason for this is to get an idea, whether the problem of dipole localization can be solved on today's machines, avoiding the use of high performance or parallel computers.

The modifications of the code were tested on the following machines:

- PC with Pentium Pro Processor running under Linux
- Compaq workstation with an Alpha 21164 running under Digital Unix 4.0
- Compaq workstation with an Alpha 21264 running under Compaq Tru64 Unix
- PC with an AMD Athlon running under Linux

The Pentium Pro and both Alpha processors were designed as high performance processors for numerical operations. The AMD Athlon is the latest processor in this group and is a mainstream device found in a lot of private PCs today. It is a low cost processor and therefore, it is quite interesting to see its performance in this project.

To get some detailed information about the processors, see appendix A.

5.2 Compilers

For both PCs, the GNU gcc compiler was deployed. This is a mainstream C-compiler available for a lot of processor types that can do some optimization. The machine code for the PCs is x86 code CISC. But, as already mentioned in section 4.1, both PC processors behave like a x86 CISC to the outside, but internally split the machine codes into RISC-like sub-codes and use all the features mentioned in chapter 4. The only problem with this is the fact, that

there are only a few registers available for the machine code language, so that the compiler cannot do a good job in optimizing the code with a sophisticated register utilization. So it is for example not possible to keep often used variables (e.g. loop counters) in the registers. Furthermore, the compiler only allows to optimize 486 code, a predecessor of both processors. The software was compiled with the following flags:

```
gcc -O2 -m486 ...
```

On the Alphas, the special compiler for the Alphas was used. This is a sophisticated highly optimizing compiler. It was used with the following flags:

```
cc -fast -arch host ...
```

This configures the compiler to produce the most efficient code on the current machine with no regard to efficient portability. The compiler is also able to use runtime information about the memory access pattern to align the data to the cache architecture. This feature was not used in this project, because it would interfere with the data alignment attempts in this work. So drawing any conclusions about modified data alignments would be difficult.

5.3 Performance Analysis Tools

In order to evaluate the influence of certain code modifications, you need some measure for performance. In this project two utilities were used:

PCL The Performance Counter Library, Version 1.3

This tool was developed at the Research Center Juelich GmbH, Germany. It provides a common interface to the hardware performance counters on various architectures. It implements some easy-to-use functions to be placed in the code, telling the tool, which measures you are interested in and which part of the code has to be monitored. This part is simply embraced by a start and stop command. The tool uses the same interface on every system in order to provide portability. But, of course, some measures are not available on some architectures. It provides various measures for

- the memory architecture (caches and TLB),
- clock cycles spent in the code,
- number of various instruction types issued,
- jumps and branch predictions,
- and atomic instructions.

Furthermore, it can compute some rates, like the MFLOPS (mega floating point operations per second) and cache miss rates.

This tool was used to measure and compare the codes. It was placed around the iteration code of the SOR to measure the performance of a single iteration step. In the end, the average value of all iterations is printed out, so that side effects during a single iteration are blurred. For more information on PCL refer to [16].

DCPI The advanced profiling tool from Digital for the Alpha workstations

This tool gives detailed information about a program's execution, shows stalls, cache misses, ... It is able to show this information for all lines of the code or the assembler code. So it is easy to identify the slowest portions of the code and the reasons for them. This tool was mainly used for the final optimization. See [17] for details on DCPI.

Chapter 6

The Original Code

This chapter introduces the original source code for solving the system of linear equations using SOR (see chapter 3). This piece of code is used in the preparation step of the whole dipole localization algorithm (see figure 2.3) in order to calculate the $m-1$ potential distributions needed for FDRM. The goal of this work has been the optimization of this piece of C-code, as it is numerically very intensive.

6.1 SOR in the Code

The formula used in the original code for the iteration of every cube is:

$$u^{new} = u^{old} + \omega \frac{residual}{cc} \tag{6.1}$$

This is basically the same as equation (3.2), but the variables have been renamed for the application. u denotes the potential of the corresponding cube. The *residual* is the difference between the right and left side of the corresponding linear equation (refer to equation (2.9)). cc is the central coefficient which is the sum of all six neighbor coefficients of the current node (see equation (2.11)).

6.2 Red/black Iteration

The code updates all grid points in a cubic workspace by using three nested loops. But it does not update every point in one run, but only half of them. This is called red/black iteration: first all red points are updated and, in a second sweep, all black points. A red point is characterized by an even sum of coordinate indices. A black point has an odd sum of coordinate indices. In a plane, this pattern looks like a chessboard, while in 3D, it resembles several chessboards, one behind the other, with alternating colors form one plane to the other. The idea behind it is the fact, that all red points have only black neighbors and vice versa. So when calculating the $i+1^{st}$ iteration, you first update all red points using the old (from iteration: i) black data of the neighbors and afterwards, you update all black points using only new data (from iteration: $i+1$) of the red points. This method needs a smaller number of iterations than updating the grid points one by one using only a single sweep through the workspace.

6.3 Data Structures

Four data structures are used for the SOR code. xc, yc and zc are three dimensional *float*-arrays holding the coefficients in x-, y- and z-direction of every cube in the model towards one neighbor (the one with the higher index) in the corresponding direction. You do not have to save the coefficients to the other neighbor explicitly, because the coefficient from one point to a neighbor is the same as the coefficient from this neighbor back to the original point. This is a result of the fact, that the conductivity from one cube to a neighbor cube is the same in both directions. u is also a three-dimensional array holding the result potentials of type *double*.

These data fields are not real 3D C-arrays but organized as pointer structures to be able to handle different data sizes. Let us pick u as an example: Look at figure 6.1 to see the internal structure. u is an array of pointers to the different planes in the workspace. Every pointer points to a $ysize$ area of pointers denoting the different rows in the plane. These pointers point to the real data values in the corresponding row. Notice, that the data is allocated in one coherent block of memory. This is important for good cache performance when working sequentially through the data.

6.4 Description of the Code

Refer to the code structure in figure 6.2 and to the code fragment in figure 6.3. The whole iteration is encapsulated in a `while`-loop that repeats the iteration until the residual norm is small enough. Then, you see three nested loops working through the whole cubic workspace of the head model. The k-loop cares for the red/black iteration by only choosing red points. The first step for every grid point is the calculation of the central coefficient (cc) which is the sum of all six neighbor coefficients. As no current can leave the head, it is important to know, whether the current point is inside the head. This is the case, if $cc! = 0$. This is checked next. The calculation proceeds only in the case of a valid head point. Then the residual is determined by adding the data of the current point and the six neighbors. The right hand side entries are not saved for every equation explicitly, because only two (the two poles of the dipole) non-zero right hand sides can occur. Therefore, an array with four *double* values per entry is defined, one entry per non-zero right-hand side. The first three *double* values are the coordinates of the pole that have to be casted to an integer for comparison (This will be removed during the optimization later). The fourth entry is the value of the dipole's current.

Afterwards, the update of the potential at the current node is performed using formula (6.1). Then, the residual, which is the difference between right and left side of the linear equation, is squared and added to the squares of all other residuals in this iteration ($resnorm$). This value is used to determine the end of the iteration.

The whole procedure is repeated for all black points.

6.5 Adaption of the Code

The code was used as an add-on to the "Matlab" software package. For the optimization, it was changed to a stand-alone application. Furthermore, the access to the arrays was changed in the way, that the inner loop (k) works through the last coordinate of the xc, yc, zc and

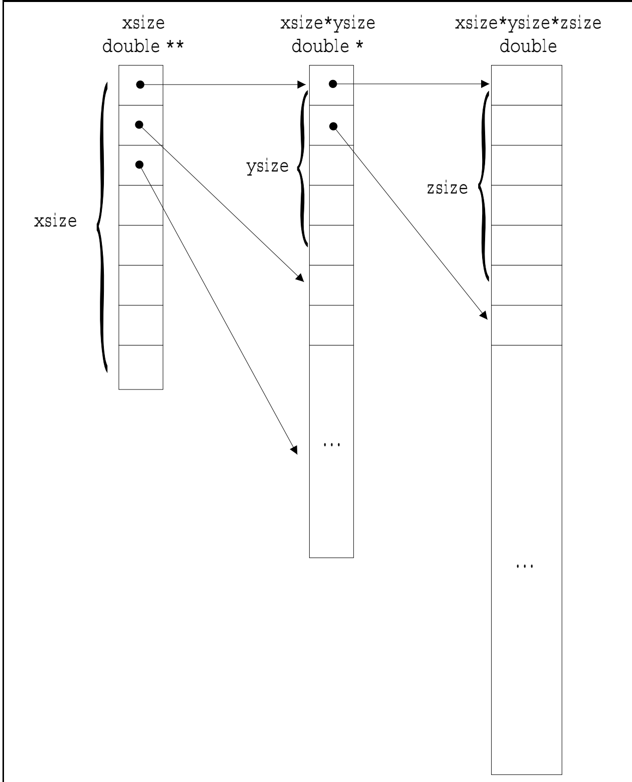


Figure 6.1: Internal array structure in the original code

u arrays. In the C-language, this means walking sequentially through the main memory and thus using the cache capabilities most efficiently. Not following this rule delays the program by approximately a factor 3.

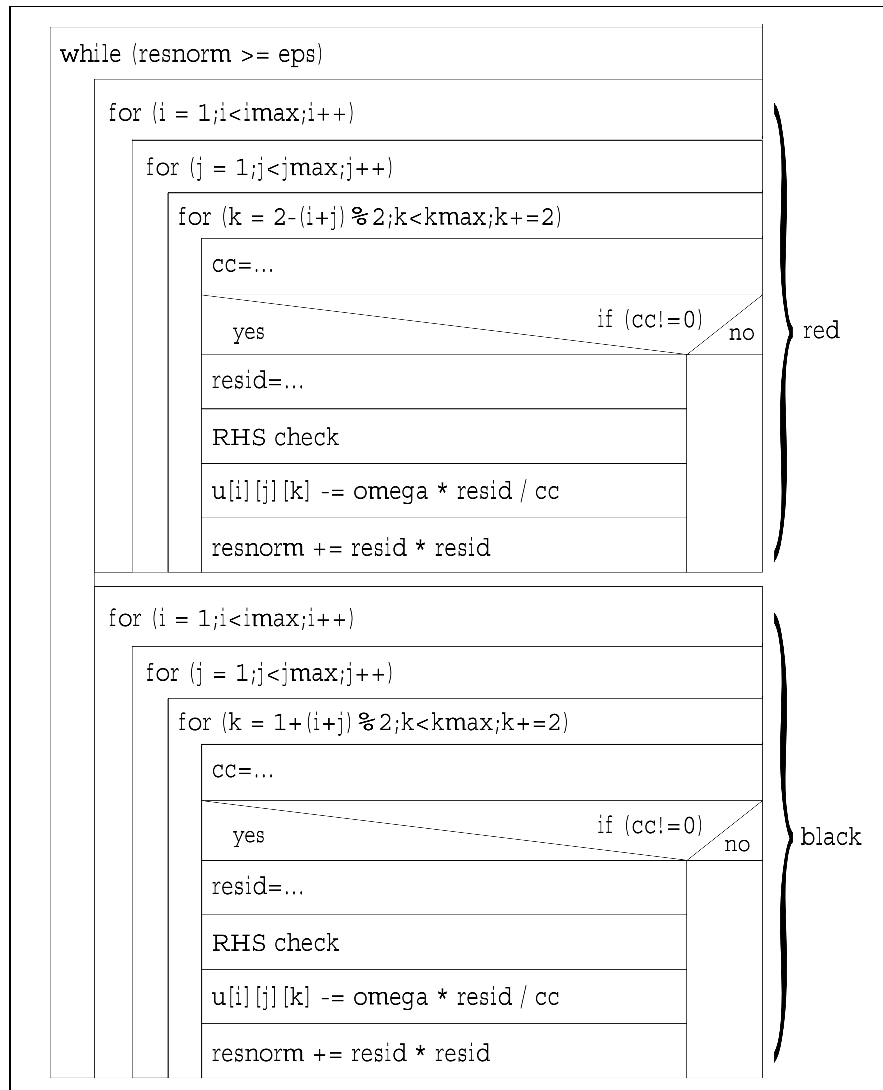


Figure 6.2: Structure of the original code

```
1  while ( n_sweeps <= maxit && resnorm >= eps )
   {
     /* Increase sweep counter */
     n_sweeps++;
5
     /* Initialize residual norm to zero */
     resnorm = 0.0;

     /*******************************************************
10    *  Sweep over the red points (sum of indices is even)  *
      *******************************************************/

     for( i = 1; i < imax; i++ )
     {
15      for( j = 1; j < jmax; j++ )
        {

           /* if the sum of the indices i+j is even, then the first red
              point that we must treat in this column has index 2, if
20            the sum is odd, the index is 1 */
           for( k = 2-(i+j)%2; k < kmax; k += 2 )
           {

              /* Determine Central Coefficient */
25            cc = - (double)( xc[i-1][ j ][ k ] + xc[i][j][k] +
                               yc[ i ][j-1][ k ] + yc[i][j][k] +
                               zc[ i ][ j ][k-1] + zc[i][j][k] );


30            /* We only have to treat this node, if the central
                 coefficient is non-zero */
              if( cc != 0.0 )
              {

35               /* First compute the part of the residual that
                    does not involve the rhs */
                 resid =
                    (double)xc[ i ][ j ][ k ] * u[i+1][ j ][ k ] +
                    (double)xc[i-1][ j ][ k ] * u[i-1][ j ][ k ] +
40                  (double)yc[ i ][ j ][ k ] * u[ i ][j+1][ k ] +
                    (double)yc[ i ][j-1][ k ] * u[ i ][j-1][ k ] +
                    (double)zc[ i ][ j ][ k ] * u[ i ][ j ][k+1] +
                    (double)zc[ i ][ j ][k-1] * u[ i ][ j ][k-1] +
                    cc * u[ i ][ j ][ k ];
45
                 /* Now add part that depends on rhs */
                 for( l = 1; l <= n_rhs; l++ )
                 {
                    if ( (int)rhs[l][1] == i &&
50                       (int)rhs[l][2] == j &&
                         (int)rhs[l][3] == k )
                    {
                        resid += rhs[l][4];
                    }
55               }

                 /* Add scaled correction to value at current node */
                 u[i][j][k] -= omega * resid / cc;

60               /* Sum up squares of residual */
                 resnorm += resid * resid;

              }
           }
65      }
     }
```

Iteration of black points

...

Figure 6.3: Red iteration part of the original code

Chapter 7

Code Optimization

This chapter describes the steps to optimize the code for solving the forward problem (see figure 6.3). Furthermore, the results of all these optimization steps are presented and unexpected results are emphasized. We try to interpret the results using the underlying computer hardware properties wherever possible. Successful modifications are kept in the code so that it is improved gradually in the course of the following sections. So the results in this chapter cannot be seen as independent steps. All measurements were done using a real head data set of 65 grid points per dimension. This is a fixed resolution of the medical device used to create this head data set.

The performance results presented are the average number of cycles, cache misses, etc. per iteration of the whole data set i.e. of one complete SOR sweep.

To present results for the Athlon, where PCL was not available, the user time dedicated to the code was measured and is presented in the tables. This time comprises the whole run of the program including the loading of the data and all preparation steps.

The Alpha 21164 is called Alpha 1 and the Alpha 21264 is denoted as Alpha 2 in the tables of this chapter.

It should be noted, that all code versions in this chapter except of the linear iteration in section 7.2 do not modify the order and algorithm of data update. So the final optimized version still computes the same results as the original code.

In order to get an overview of possible optimization steps in numerical codes in general see [5], [6]. For more specific Alpha processor optimization refer to [18], [19], [20] and [21].

7.1 Limiting the Processing to Head Points

The first objective was to improve the algorithm with respect to its working set: Only grid points inside the head (so called *head points*) need to be updated. The whole working set is a cube, but the head has the shape of a sphere, which only fills a part of this cube.

7.1.1 Precalculation of the Central Coefficient

The central coefficient (cc) of every grid point is used to decide whether the corresponding point is inside the head ($cc! = 0$). So it determines whether an update of the potential has to be done. This central coefficient is equal to the sum of all six neighbor coefficients and is calculated in the original code (figure 6.3) in lines 25-27 (notice, that it is also calculated in

the black iteration part in a similar way). It is computed for every point in every iteration. As this value is constant for every grid point, the first idea was to precalculate it:

Before the iteration part of the code starts, an additional double precision 3D array is allocated in the same manner as figure 6.1 and filled with the values of the corresponding *cc*.

In case of a head point we simply save the calculation of *cc*, but have to transfer an additional value to the processor which might lead to a cache miss. Notice that the six neighbor coefficients are needed anyway in line 37 in this case. The big advantage of the precalculation becomes clear when thinking of a point not in the head: This only leads to one memory access (to the *cc* array) without floating point calculations, instead of six memory accesses and the subsequent floating point additions in the original case.

7.1.2 Dynamic Loop Boundaries

Another idea to restrain the calculation to grid points inside the head is to simply avoid touching grid points outside. This is done by allocating two 2D-arrays of integer values and precalculating the start- and end-index in z-dimension (for the innermost loop) for every pair of x- and y-coordinates. The start-index represents the first z-coordinate of a point of the head for the given x- and y-coordinate. The end-index represents the last one. The k-loop is modified to run from the corresponding start- to the end-index given the current x- and y-coordinate. So we have dynamic loop boundaries in the innermost loop. If there are no valid grid points, the corresponding start-index is simply set to a larger value than the end-index so that the loop body is not processed. Notice, that it is not possible to leave out the `if`-command in line 32 because parts of the head are concave (e.g. the eye socket) which will result in grid points with *cc* equal to zero but inside the interval covered by the start- and end-index.

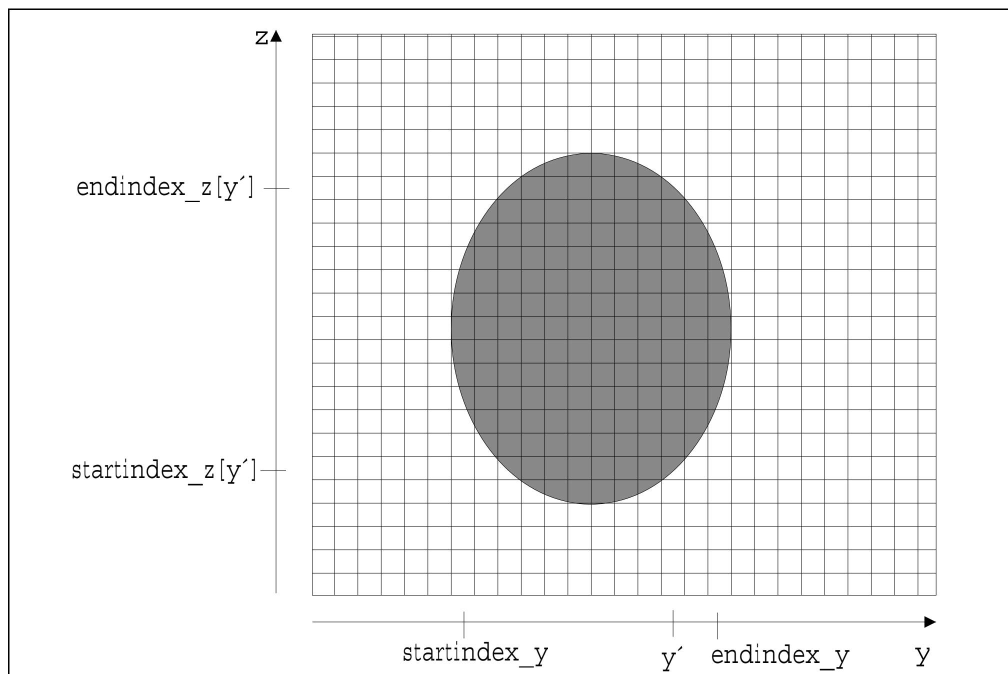


Figure 7.1: Visualization of dynamic loop boundaries (2D)

Furthermore, it is possible to adopt this scheme to the y-direction, too. It is possible to determine the valid y-coordinate interval for a given x-coordinate using the start- and end-index information for the z-direction. The y-start-index shows the first column in which valid z-coordinates exist. So it is also possible to use dynamic loop boundaries for the j-loop. Notice, that this does not reduce the number of inner loops processed but helps to reduce unnecessary jumps of the program counter.

Figure 7.1 visualizes the idea of dynamic loop boundaries in a 2D case. You can determine the start- and end-index in z-direction for a given y'. Furthermore, you can limit the y-interval that has to be processed.

An extension of this idea to the x-direction was not done, because the i-loop is only processed once per iteration and color. So it will not speed up the code noticeably.

7.1.3 Results

Table 7.1 shows some PCL measurements of different versions of the code deploying *cc* precalculation and/or dynamic loop boundaries. First of all, look at the cache behavior (L1DCM=Level 1 Data Cache Miss, L2CM=Level 2 Cache Miss, L2CRW=Level 2 Cache Read/Write). The precalculation of the central coefficient leads to a higher cache performance because grid points outside the head only access the new *cc* data instead of all six neighbors. Furthermore, it reduces the number of floating point instructions (FPInst) because no computation is necessary for points outside the head. Using dynamic loop boundaries improves the cache performance again because only points within the convex hull of the head are touched at all. Using dynamic loop bounds also in y-direction does not lead to a significant gain. This is due to the fact that the number of inner loops processed is not reduced by this technique. It only avoids situations where the program runs in a k-loop with an end-index less than the start-index (i.e. nothing to do).

At the first glance it might be astonishing that the version using only dynamic loop boundaries and no *cc* precalculation leads to best cache and overall performance. Due to dynamic loop boundaries, this touches nearly only points in the head, except of the ones in the concave parts of it. The coefficients needed to calculate *cc* are needed later on for the update of the potential. So this is not a useless transfer for head-points. But the additional transfer of a precalculated *cc* is omitted, saving cache misses. The only disadvantage is that *cc* has to be calculated every time, leading to a higher number of floating point instructions. Obviously, this is better than additional cache misses. This effect can also be seen on the other architectures (see figure 7.2 and table 7.2). Note that PCL is not available for the Athlon. So only the time for the whole forward problem solution is regarded. Table 7.2 also shows the ratio of the dynamic loop boundaries version and the original version.

The dynamic loop boundaries also reduce the number of mispredicted branches (JPUS=jump unsuccess). This is obvious because nearly all `if`-commands in line 32 are successful. So pipeline flushes become less probable and the internal parallelism of the processors is deployed more efficiently. Dynamic loop bounds also reduce the total number of jumps. This, in turn, leads to a lower absolute number of successfully predicted jumps (JPSU).

Figure 7.3 shows to what extent the number of inner loops entered can be reduced by dynamic loop bounds in the real head model. In addition, the overall processing times, including the loading and preparation times of the data set, on the Pentium Pro are displayed for the different versions.

Version	Cycles (millions)	FPInst (millions)	JPSU (thousand)	JPUS (thousand)
Original Version	33.51	2.92	719.1	17.8
cc precalculation	29.35	1.67	718.2	17.6
cc precalculation + dynamic loop bounds (z)	23.57	1.49	368.5	5.7
cc precalculation + dynamic loop bounds (y,z)	23.48	1.49	360.4	5.5
Dynamic loop bounds (y,z)	21.89	1.85	360.3	5.5

Version	L1DCM (thousand)	L2CM (thousand)	L2CRW (thousand)
Original Version	693.8	693.8	737.2
cc precalculation	558.3	250.3	563.5
cc precalculation + dynamic loop bounds (z)	459.6	158.4	466.4
cc precalculation + dynamic loop bounds (y,z)	456.6	157.5	463.6
Dynamic loop bounds (y,z)	359.1	118.7	377.9

Table 7.1: Specific results for cc precalculation and dynamic loop boundaries on the Pentium Pro

Table 7.3 examines the influence of the orientation of the head with dynamic loop boundaries. The head is turned so that the original dimension in the left column of the table becomes the new z-direction. Then, dynamic loop boundaries are determined and the number of points to be processed are computed (right column). To see the influence of the direction imagine the eye socket: It is omitted from the iteration when the z-axis is directed from the front to the back of the head.

As the number of processed points does not vary significantly, a transformation of the head model is not necessary.

Table 7.4 shows the number of cycles for the overhead of all code versions. This is always smaller than a single iteration. So it can be neglected in either case.

As a result, it can again be seen that the version deploying only dynamic loop boundaries is the fastest one. So it seems to be useful to adopt the algorithm to the specific geometry of the working set of the problem in this project even if this causes some overhead. This version is used for further optimization in the following sections.

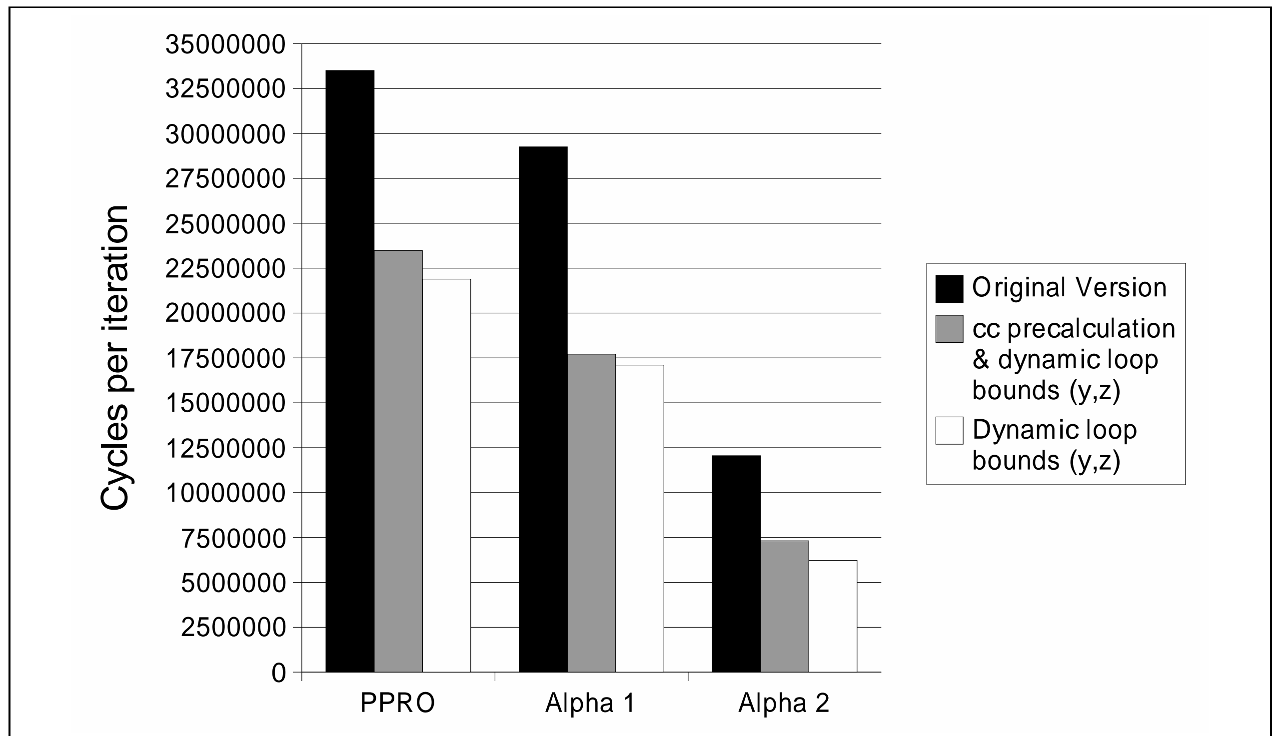


Figure 7.2: cc precalculation and dynamic loop boundaries on different architectures

Version	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
Original Version	33.51	29.26	12.05	4.15
cc precalculation + dynamic loop bounds (y,z)	23.48	17.71	7.31	2.78
Dynamic loop bounds (y,z)	21.89	17.10	6.22	2.51
Ratio (dynamic/original)	0.65	0.58	0.52	0.60

Table 7.2: Cycles per iteration for cc precalculation and dynamic loop boundaries on different architectures

Fine dimension	Number of grid elements to process
z-dim	71600
y-dim	71964
x-dim	72489

Table 7.3: Number of grid points to process applying dynamic loop boundaries with different orientations

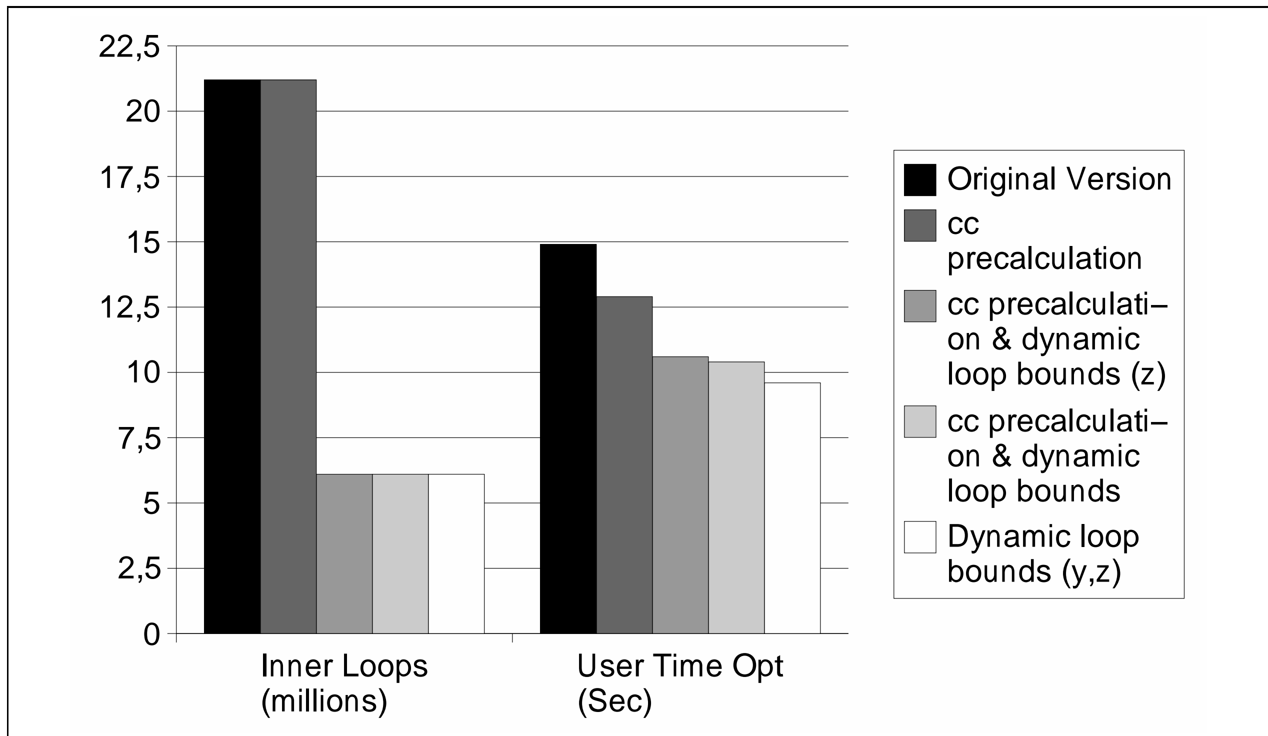


Figure 7.3: Inner loops and overall processing time for the cc precalculation and the dynamic loop boundaries version

Version	Cycles (millions)
Original Version	0
cc precalculation	12.51
cc precalculation + dynamic loop bounds (z)	15.47
cc precalculation + dynamic loop bounds (y,z)	15.35
Dynamic loop bounds (y,z)	10.20

Table 7.4: Preparation costs for cc precalculation and dynamic loop boundaries on the PPRO

7.2 Linear versus Red/Black Iteration

The original algorithm uses a red/black pattern for updating all grid points (see section 6.2). This procedure works twice through the whole data set. This roughly leads to a double amount of transfers to the cache in comparison to a linear update scheme. This section examines whether the performance gain achievable by reduced cache misses of a linear iteration can cope with the reduced number of iterations of the red/black method.

A linear update and a red/black update are compared. Figure 7.4 shows the ratios of various quantities between linear and red/black ordered SOR on the Pentium Pro. The level 2 cache misses are indeed reduced to nearly 50% because of the single sweep through the data in a linear iteration. But the algorithm touches all six neighbors of a point in an update. The two points in x-direction have a distance of $\pm 65^2 = \pm 4225$ entries in memory. Obviously, the level 1 cache cannot store data of all three planes needed for an update so that its behavior does not improve that much. But the level 1 cache is more significant due to its speed.

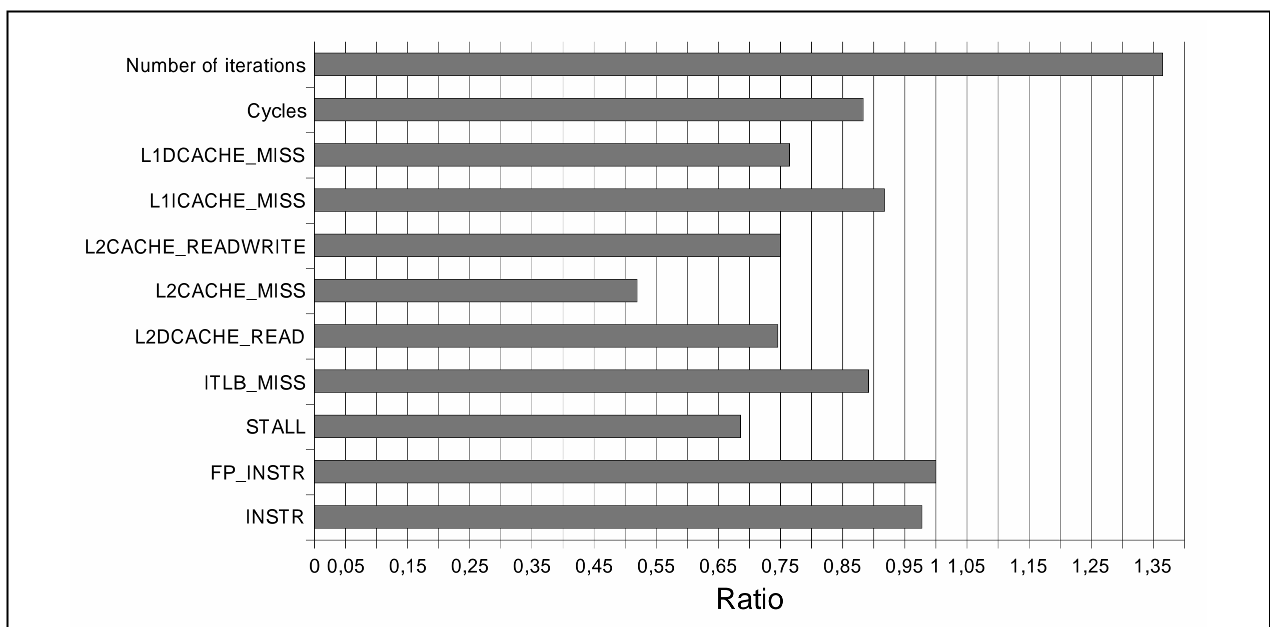


Figure 7.4: Linear versus red/black iteration

The number of cycles units have to wait for other units or the memory system (STALL) is reduced by 30%. But the overall performance of an iteration does not improve dramatically (Cycles). Only about 10% of the cycles are saved in a linear update. The number of iterations needed to achieve the given precision in this test is much lower in red/black. It was not possible to adopt the manually determined relaxation factor for the linear iteration to cope with red/black. The cache gain of the linear method will be even smaller in future because the data set size will be larger for the real clinical application. This will lead to a smaller gain in level 2 cache behavior, because it will not be able to keep data of neighboring planes. So further optimizations will keep the concept of red/black iteration.

Refer to chapter 8 for a technique that avoids two sweeps through the data set but keeps the red/black scheme.

7.3 The Influence of Casts

7.3.1 Right-Hand Side Casts

As already mentioned in chapter 6 the right-hand side (RHS) in the application does not vanish at two single points. These are the positions of the clamps that are regarded in the calculation. A RHS entry is represented by four double precision values. The first three entries represent the coordinates of the pole, the fourth one is the current value. The number of possible non-zero RHSs is not limited in the code. Have a look at lines 47+ in figure 6.3. The coordinates of the RHS structure are always converted to integers and compared to the loop indices. The next optimization was to store the RHSs in a structure consisting of three integer coordinates and one *double* value. So the conversion can be omitted. The representation of a floating point value in contemporary processors is totally different than the format of an integer. Integers are stored directly as binary numbers. But floating point values have a different format consisting of a sign, a mantissa and an exponent. The conversion is not trivial and has to be done at runtime.

See table 7.5 for the PCL results on the Pentium Pro. The new structure leads to a 40% gain in time. About 15% less instructions have to be carried out in an iteration. The number of stalls increases a little bit. Figure 7.5 shows the results for all processors. The Alphas have a machine instruction to transfer data from an integer to a floating point register ([12]). So they support the conversion in hardware, which leads to a reduced gain of the new RHS representation.

	Time (sec)	Cycles (millions)	INSTR (millions)	STALL (millions)
With casts	9.5	21.88	9.21	7.74
Avoid cast	5.7	13.03	7.94	8.27
Without/with	0.60	0.60	0.86	1.07

Table 7.5: Influence of RHS casts on the Pentium Pro

The new representation speeds up the PC processors significantly. Therefore this representation will be kept in the following sections.

7.3.2 Standardizing Data Types

Another idea was to use as few data types as possible in the algorithm to avoid casts. The original algorithm uses double precision numbers for the result u and for the right-hand side. Furthermore, it uses single precision values for the coefficients xc, yc and zc. Two versions of the code were generated: One using only single precision variables (all-float) and one using only double precision variables (all-double). You have to keep in mind that in the C programming language all floating point calculations are processed with double precision internally. This cast cannot be avoided.

See table 7.6 for the results: The all-float version is faster on most machines. This comes along with the reduced memory demand of the u-values (now *floats*) and a reduced number of cache misses. The Pentium Pro is slowed down by the all-float version what might be

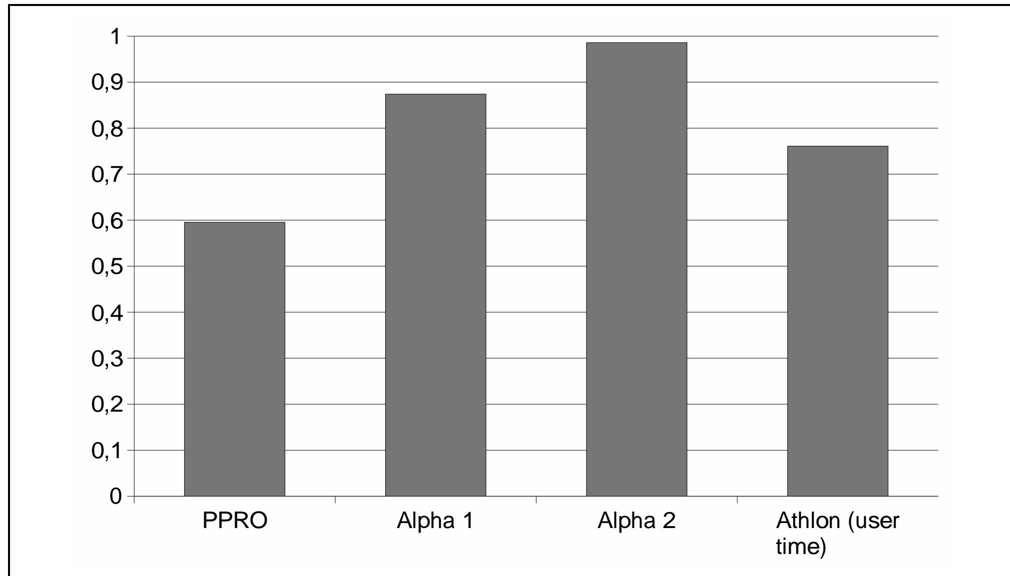


Figure 7.5: Cycles per iteration ratio avoiding the RHS casts

caused by a slower single/double precision conversion for the actual calculation or some cache alignment effects.

	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
Reference	13.03	14.95	6.14	1.91
all_float	13.89	13.08	4.57	1.67
all_double	15.19	16.47	7.86	2.17

Table 7.6: Cycles per iteration using only one floating point data type on different architectures

The all-double version is slower on all processors. The memory usage for the coefficients is twice as high in this case, leading to a higher number of cache misses and transferred bytes.

The all-float version is faster on the Alphas and on the Athlon but has a reduced precision. It was not yet analyzed whether single precision potentials lead to a satisfying dipole localization accuracy. Therefore, the data structures are kept in the way they were in the original version.

7.4 The Inner `If`-command

7.4.1 Costs of the Inner `If`

In this section we examine the costs of the inner `if`-command (line 32 of the code figure 6.3). We want to decide whether some overhead can be tolerated in order to remove this branch condition. As already mentioned in section 7.1, the central `if`-command cannot be left out because there might be some grid points covered by dynamic loop boundaries but not inside

the head. These are only a few points. So the branch is entered very often and the dynamic branch prediction works well. In order to determine the costs of the branch, the head data set was changed. The new set has only valid points in the intervals of the dynamic loop boundaries. So the branch can be left away.

Figure 7.6 shows the results. A significant gain can only be found on the Alpha 21164. All the other processors seem to have no problems with branches of nearly 100% success. The reason for the loss of performance on the Pentium Pro could not be found and might be caused by some side effects.

7.4.2 Avoiding the Inner `If`

In order to avoid this inner condition check, the algorithm has to make sure that the inner loop body is only entered in case of a head point. This is feasible by not only defining one start- and end-index in z-direction for the first and last head point, but as many as needed. So a bunch of intervals (so called *clusters*) is defined for every pair of x- and y-coordinates. A cluster is a coherent range of head points. In order to implement this, an additional loop level has to be introduced in front of the k-loop. This new loop iterates through the number of clusters for a given x- and y-coordinate. The inner loop runs from the start-index to the end-index of the current cluster. Figure 7.7 visualizes the idea in 2D: For a given y-coordinate (y') the cluster data structure stores the start- and the end-indices of valid points. The boundaries are stored in an array b here, just like in the real code.

Figure 7.8 shows the performance of the cluster-based version: The overhead of managing clusters destroys the gain of the omitted `if` on the Alpha 21164. The Alpha 21264 is the only architecture that can improve the code performance by roughly 10%. The cluster-based version is not used in the following sections, but the idea will be deployed again in the 1D representation (section 7.10).

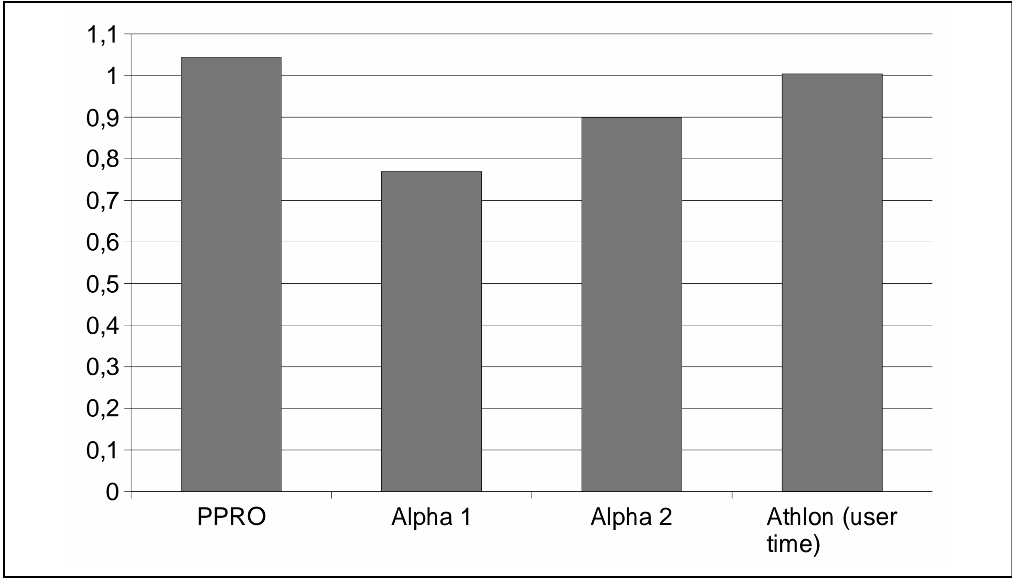


Figure 7.6: Cycles per iteration ratio avoiding the inner `if` on different architectures

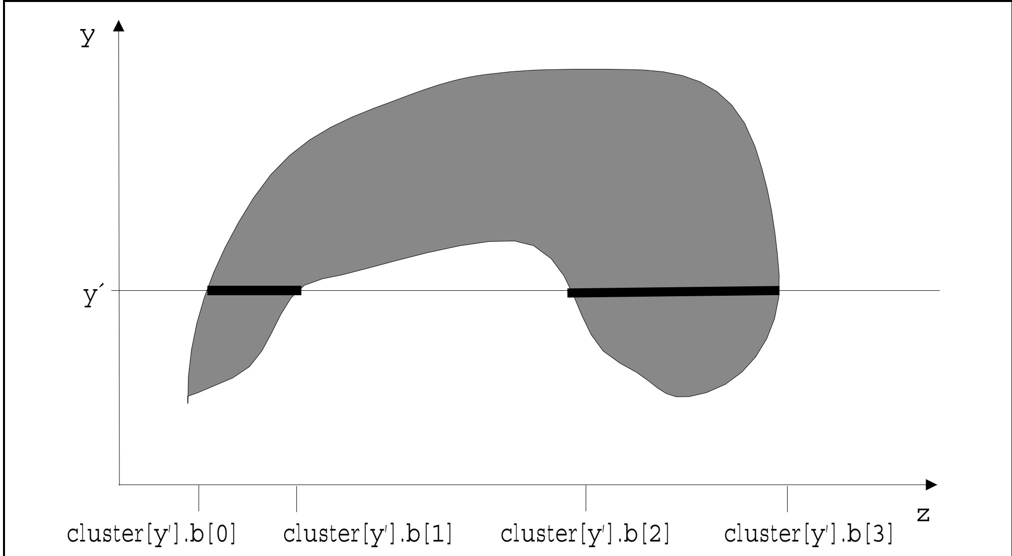


Figure 7.7: Visualization of the cluster method

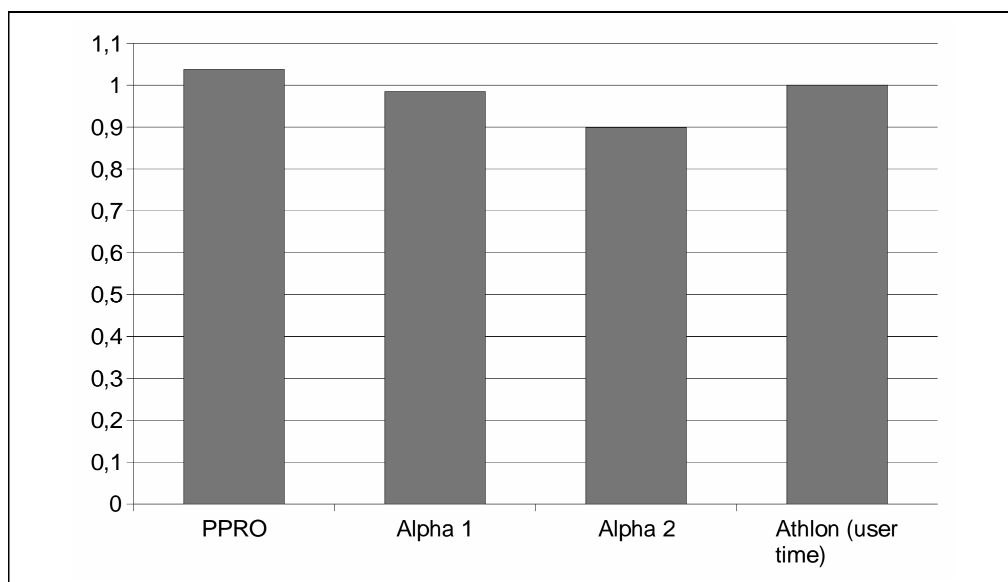


Figure 7.8: Cycles per iteration ratio of the cluster-based version on different architectures

7.5 Influence of the RHS Check

This section examines the costs of the right-hand side (RHS) check in the inner loop (lines 47+ in figure 6.3). This piece of code checks whether the current coordinate is a pole of the dipole or, in the way the code is used in this project, whether it is a clamp position of the current clamp pair. Clamps can only be located on the head surface. Therefore, it is possible to check the RHS only on the margin of the head. The code could be rewritten in a way that the first and last point of a z-interval (i.e. at the dynamic loop boundaries) are treated in the original way while all points in between can omit the RHS check because they are inside the head.

A code version without an RHS check was written and some initial values of u were set, avoiding an all-zero initial condition. This version was compared to the one with RHS check. The ratio is presented in figure 7.9. The speedup is about 5% to 14%. One could think that it should be higher, but the integer coordinates comparisons of the RHS check will be carried out simultaneously to some floating point calculations in superscalar processors. The only part that depends on the RHS is the floating point addition in line 53.

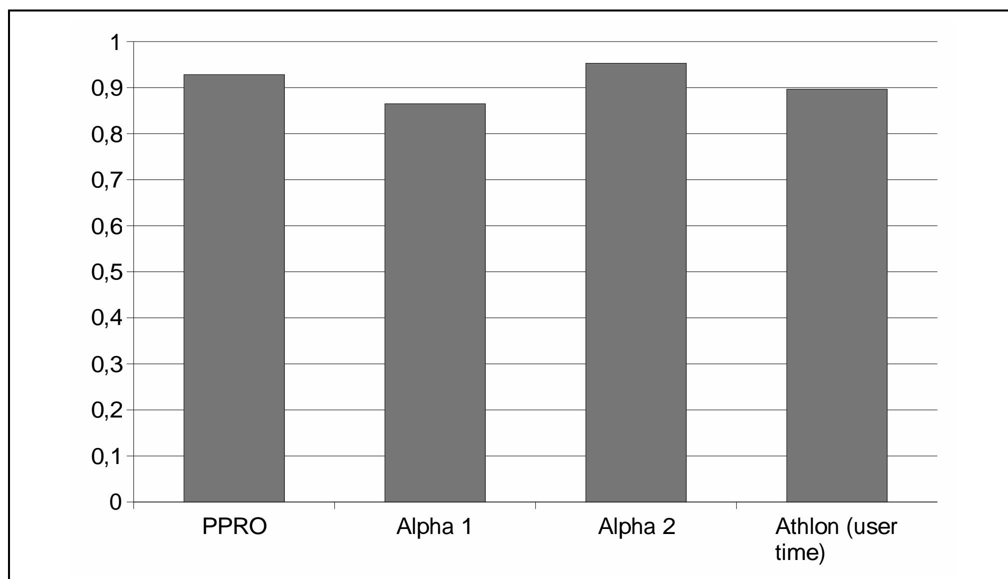


Figure 7.9: Cycles per iteration ratio without the RHS check on different architectures

The gain might be compensated by the overhead of a separate treatment of the margin points. As this modification of the code would reduce its flexibility, no change of the RHS part was done.

7.6 Static Array Sizes

A well-known principle in code optimization is to tell the compiler as much as possible about the problem the code solves. This helps the compiler to optimize code most efficiently. So pointers are not good in a code because it might not be clear to the compiler, whether some other variables can be modified through the pointer. So it has to be very restrictive in optimizing and software-pipelining in order to assure a correct execution [6]. In post-RISC

processors (see section 4.8) this problem is not so evident because the processor can decide at runtime whether an operation affects other ones so that these instructions have to be serialized.

This section examines the behavior of the code using static array sizes. This means all data structure sizes are hard coded for the 65^3 grid point data. The 3D arrays are also declared as 65^3 standard C-arrays avoiding the indirection shown in figure 6.1. So accessing a specific grid point is simply an integer index calculation and one memory access. But static array sizes reduce the flexibility of the code.

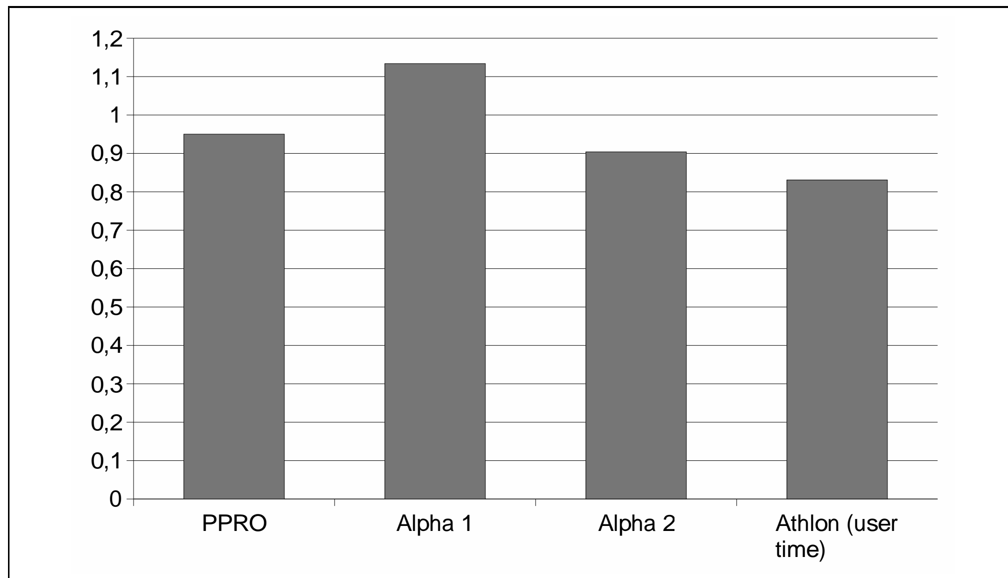


Figure 7.10: Cycles per iteration ratio using static data structures on different architectures

The results are shown in figure 7.10. The Alpha 21164 even slows down in this case. This could be caused by some cache conflicts and solved with padding techniques (see section 7.8), but this was not analyzed in the scope of this work. The highest gain can be seen on the Athlon (about 15%).

The idea of avoiding the threefold pointer dereference (figure 6.1) for accessing grid data will be pursued in the 1D representation later on (section 7.10). Due to the loss of flexibility, the code will not be changed for further optimization.

The Alpha compiler provides a flag to indicate that pointer variables are *restricted*, meaning they will not be used to influence other variables. There is no significant gain on the 21264 using this flag. This underlines the assumption that post-RISC processors get their performance mainly from the runtime information and out-of-order execution instead of sophisticated compiler techniques in this case.

7.7 Different Data Arrangements

This section compares three different data arrangements:

- The arrangement of the original code: Keeping *xc*, *yc*, *zc* and *u* in different data structures and, by this, in separate memory blocks.

- See figure 7.11(a). All coefficients (northern, southern, western, eastern, upper, lower and central coefficient) for one point are put in a structure, *u* stays separate: This arrangement does not use the symmetry of the conductivities. So the memory usage is nearly twice as high. The main drawback of this arrangement is the fact, that for an update of a value seven of these structures have to be touched and loaded into the cache, although only one is needed completely. Only the *u* members of the six neighbors are needed. The structures are allocated and accessed in the same way as the coefficients in the original code (figure 6.1).

- See figure 7.11(b). Three coefficients of a point are put together: This arrangement puts *xc*, *yc* and *zc* of one point in neighboring memory locations. *u* stays separate. The amount of data stays the same in this case.

The last two arrangements are called *array merging*.

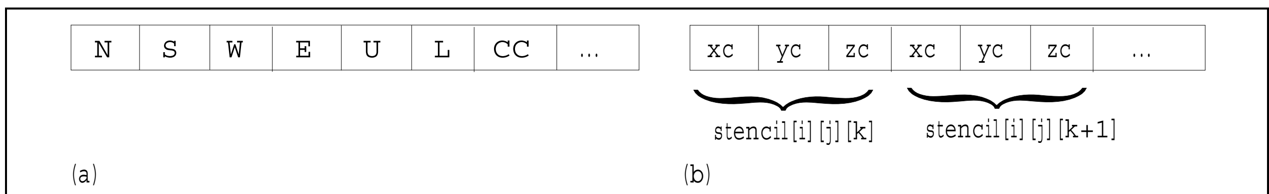


Figure 7.11: Different data arrangements

The results can be seen in table 7.7. Putting all values in one structure slows down most of the architectures because of the higher memory usage and the inefficient cache policy. The last arrangement improves the performance on all architectures except the Athlon. As PCL is presently not available for this processor, it is not possible to find the reason for this. The idea of this data arrangement is also used in the 1D representation later in this chapter. There, a cache performance analysis will be presented for it (see table 7.11).

	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
Original structure	13.03	14.95	6.14	5.91
All values in a structure	13.22	14.74	6.70	7.49
Three coefficients in a structure	12.38	13.54	4.78	7.52

Table 7.7: Cycles per iteration using different data arrangements

7.8 Padding

The technique of adding unused elements to a data structure/array or to change the memory base address of structures/array in order to avoid cache conflicts is called *padding* (see [8] for details). The objective is to avoid that the data of different structures currently processed is put into the same cache line by hazard and leading to an inefficient use of the cache. The code version with the three merged coefficients (refer to the last section) was padded by enlarging the working set in z-direction by different numbers of unused elements. No significant changes in code efficiency could be observed on all architectures. Therefore, no padding is applied in the following versions. When adapting the code to a specific processor for a clinical application, an additional padding analysis might lead to further improvement of the code performance.

7.9 Changes in the Inner Loop

This section comprises several optimization attempts for the body of the inner loop. Table 7.8 shows the results.

7.9.1 Using Additional Pointers

The first idea is to precalculate pointers to the xc, yc, zc and u data before the inner loop:

```
xc1=xc[i-1][j];
xc2=xc[i][j];
yc1=yc[i][j-1];
yc2=yc[i][j];
zc1=zc[i][j];
ux1=u[i+1][ j ];
ux2=u[i-1][ j ];
uy1=u[ i ][j+1];
uy2=u[ i ][j-1];
uz=u[ i ][ j ];
```

Within the inner loop, only these pointers are used as 1D arrays avoiding the threefold dereference of the original code. E.g.:

```
resid =
(double)xc2[ k ] * ux1[ k ] +
(double)xc1[ k ] * ux2[ k ] +
(double)yc2[ k ] * uy1[ k ] +
(double)yc1[ k ] * uy2[ k ] +
(double)zc1[ k ] * uz[k+1] +
(double)zc1[k-1] * uz[k-1] +
cc * uz[ k ];
```

This speeds up the code on every architecture. This idea is the basic improvement of the 1D representation (section 7.10).

7.9.2 Moving the RHS Part

Another possible modification of the inner loop is to move the RHS part to another position, namely in front of the residual computation. These two parts determine the residual. The modified RHS position slows down the code. So the position of the various code parts of the inner loop is important even in post-RISC architectures.

7.9.3 Moving the Central `If`

Another possibility is to move the `if(cc!=0)` to the line where a point outside the head would cause trouble, i.e. line 58 where *cc* is in the denominator. This means only this line is omitted if the current point is outside the head. This change leads to different results on different architectures. The Pentium Pro and the Alpha 21164 show no change. The Alpha 21264 is faster and the Athlon slows down.

	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
Reference	13.03	14.95	6.14	5.91
Precalc pointer to column	11.56	13.59	5.69	4.77
RHS before resid calculation	17.23	14.95	6.77	6.36
If at update of u	13.04	14.95	5.72	6.13

Table 7.8: Cycles per iteration with different modifications of the inner loop

7.10 1D Representation of the Problem

7.10.1 Basic Idea

In this section a totally different approach is considered. The results so far lead to the idea that simplifying the code (e.g. avoiding the threefold pointer dereference of the coefficients and result data structures) can improve its performance again. The design goal of the new code was to make it as easy as possible, so that the compiler can optimize it most efficiently without changing the semantics of the code. The basic concept of the new code version is the idea of completely giving up the 3D representation, i.e. using an explicit representation of the cubic working area. The data of the problem is stored in reality in one big memory block for each of xc, yc, zc and u (see right side of figure 6.1) in a coherent linear way. Looking at the data access scheme of the algorithm makes the idea clear: The algorithm works linearly through a set of data and updates every point ($u[i]$) by using six other points. These six other point have a fixed distance to the current point in the array. These are the direct neighbors (coordinate $i \pm 1$) in z-direction in the head, the two neighbors in y-direction (coordinate $i \pm zsize$) and the two points from the neighboring planes of the head (coordinate $i \pm ysize \times zsize$). Figure 7.12 shows this scheme with the sizes of the data set used in the project. The idea is to regard xc, yc, zc and u as 1D arrays and work with a 1D index. This avoids the pointer structure needed for accessing a point with its three coordinates (figure 6.1). The 1D array index is called *1D coordinate*.

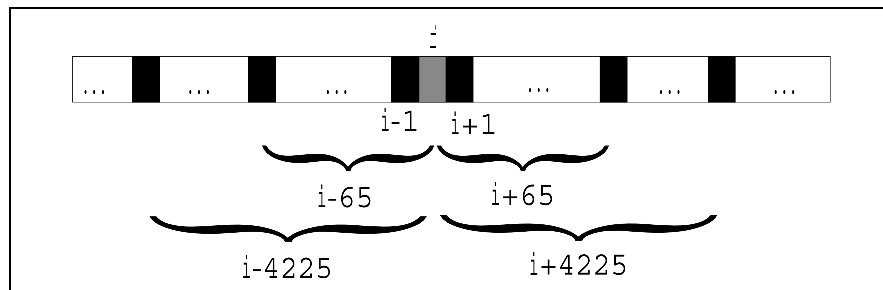


Figure 7.12: Data access pattern when updating point i using a 1D representation

Now it is possible to use only one loop running from 0 to $xsize \times ysize \times zsize$ to work through the data instead of three loops (one per dimension). Furthermore, the right hand side check can be simplified by precalculating the 1D coordinate of the pole:

```
rhs_new[l].coord=((int)rhs[l][1])*zsize*ysize+
                 ((int)rhs[l][2])*zsize+
                  (int)rhs[l][3];
```

This coordinate can be compared to the loop counter in the iteration. So only one integer comparison is needed instead of three. The following sections find the best design principles for the 1D code. They use a linear iteration (instead of red/black) for simplicity. Then, a red/black version is built (section 7.10.4).

7.10.2 Checking for Margin Points

The new representation of the problem has one drawback: It is not obvious whether the current point is on the margin of the whole 3D data set. The margin points have to be omitted from the update because the complete set of neighbor information is not available for these points. In the original version, this problem was easy to solve by simply running the i, j and k loop from 1 to MAX−1 in every dimension. Three possible solutions to this were compared. The results can be seen in table 7.9.

Version	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
Explicit cube margin determination	34.42	84.75	38.19	18.54
0/1 array	11.74	12.39	5.92	5.69
Cluster-based version	10.08	9.19	4.17	3.54

Table 7.9: Cycles per iteration using different margin detection methods for the 1D representation

Explicit cube margin determination: This method calculates the original three coordinates using the loop counter. This requires integer divisions and modulo arithmetic, which make this code too slow. This idea is not considered any further.

The next two ideas use a precalculation step which works through the cube using 3D coordinates (i.e. three loops). The first version allocates a further 1D array of the size of working area for flags. It sets the flag, if the corresponding point has to updated, i.e. the point is not on the margin and cc is not zero. It does not set the flag, if the point may not be updated. An additional condition check for the flag is necessary in the loop body of the iteration in this case.

The second version uses clusters in the way they were introduced in section 7.4.2 but now working on 1D coordinates. So only one cluster structure is allocated. The indices stored in the cluster structure show the intervals of 1D coordinates that contain points to be updated (i.e. non-margin points with cc unequal to zero). See figure 7.13: The black areas are parts of the data that have to be updated. The iteration itself consists of two loop levels now: The outer level runs through the number of clusters in the data set and the inner loop runs from the start- to the end-index of the current cluster.

So the cluster-based version needs an additional loop level, whereas the version with the flag array has an `if`-condition in the loop.

The cluster-based version is the fastest on every processor because the memory overhead for clusters is smaller than the flag array. So the cluster-based version is kept for further optimization.

7.10.3 Improving a Linear Iteration

This section describes several ideas using a linear iteration with a 1D representation. The results are presented in table 7.10. The reference for these tests was the cluster-based version of the 1D code of the previous subsection. First, different positions of the RHS check were regarded. The reference still uses the position of the original code (i.e. right in front of the actual update of u). Checking the RHS right at the beginning of the inner loop (*RHS first*

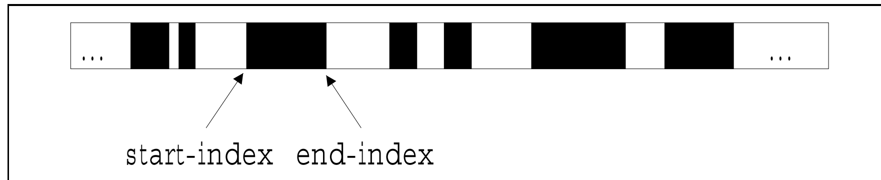


Figure 7.13: Clusters in the 1D case

in the table) leads to a loss of performance, although this should not have an influence on post-RISC machines. Checking the RHS right after the *cc* arithmetic has different results on different architectures. So the RHS check was not repositioned.

The use of pointers to the current data instead of 1D arrays with indices leads to a poor result. The data was accessed by dereferencing pointers, e.g.:

```
resid =
      (double) *xPtr1  * *Wu +
      (double) *xPtr2  * *Eu +
      (double) *yPtr1  * *Su +
      (double) *yPtr2  * *Nu +
      (double) *zPtr1  * *Uu +
      (double) *zPtr2  * *Du +
      cc * *Tu;
```

At the end of the loop all pointers were increased.

The idea of array merging by putting the three coefficients of a point side-by-side (see figure 7.11(b)) improves the performance on all processors except of the Athlon. This data arrangement is kept.

Version	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
Reference (cluster 1D)	10.08	9.19	4.17	3.54
RHS first	13.21	9.98	4.66	4.22
RHS after cc	9.88	11.51	4.52	3.92
Pointers instead of indices	12.35	9.35	4.23	5.00
Three stencils in one structure	9.20	8.81	3.64	3.99

Table 7.10: Cycles per iteration with different modifications of the 1D cluster-based version

Table 7.11 compares the cache behavior of this data arrangement to the original one for the Pentium Pro based Linux machine. Using the modified arrangement reduces the level 1 data cache misses (L1DCM) to nearly 60%. The requests to the level 2 cache (L2CRW) are reduced in the same way. Moreover, the level 2 cache misses (L2CM) can also be reduced slightly.

Some additional program versions were written on the basis of the 1D cluster-based version with the modified data arrangement. See table 7.12. The RHS check was removed in order to see its influence on this code (compare section 7.5). Another version was written that

	L1CM (thousand)	L2CRW (thousand)	L2CM (thousand)
Original data arrangement	216.9	227.4	61.8
Three coefficients in one structure	121.7	124.2	58.6

Table 7.11: Cache behavior for data arrangements on the Pentium Pro

precalculates the central coefficient (*cc*) in the way already tested with the 3D version (section 7.1). This slows down most architectures for the same reasons explained in section 7.1.

	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
Reference	9.20	8.81	3.64	3.99
No RHS check	8.46	7.45	3.15	3.61
Precalculate cc	9.03	9.62	4.03	4.18

Table 7.12: Cycles per iteration of different versions of the 1D cluster code with modified data arrangement

7.10.4 Applying 1D Representation to Red/Black

In this section a red/black iteration scheme is added to the 1D representation version and compared to the former 3D version. The results are shown in table 7.13. The reference is the 3D version using dynamic loop boundaries in y- and z-direction and the modified RHS structure to avoid casts in the inner loop.

The 1D cluster-based version is faster on all processors. When using the modified data arrangement of putting the three coefficients of a point together in memory, the 1D version is also faster on all processors.

The last entry in the table is a version that separates the data for black and red points. It defines four variables (*u_black*, *u_red*, *stencils_red* and *stencils_black*). The idea is to perform only read accesses on the red u data and only write accesses on the black u data in the black part of the iteration and the other way round in the red part. By this, every value in the memory block of the black part of u is modified, whereas the red memory block is unchanged. This should improve the performance of systems with write-back caches, because complete cache lines are modified and therefore a complete line of new data is stored to memory in case of a cache line replacement. But the overhead that comes along with the management of the data structures and indices compensates the gain.

The 1D representation is about 15% faster than the 3D code. The code using 1D representation with clusters and the modified data arrangement (three coefficients put together) is used for further optimization. Figure 7.14 shows the red part of the code version that will be the basis in the following sections.

	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
3D reference	13.03	14.95	6.14	5.91
3D three coeff. together	12.38	13.54	4.78	7.52
1D cluster-based version	11.32	11.37	5.32	4.74
1D cluster, three coeff. together	10.33	9.82	4.18	5.12
1D cluster, three coeff. together (2)	11.64	9.42	4.14	5.02

Table 7.13: Cycles per iteration of 3D and 1D versions of the code

7.11 Using Indices for Coefficients

In the head data used in the project only a limited number of conductivities can occur. Only four different compartments are distinguished: air, scalp, skull and brain. The grid points of each compartment are assigned the same conductivity. This leads to a limited number of possible coefficients (see equation (2.10) for the coefficient calculation): $\binom{4}{2} = 6$ that do not necessarily have to occur all in the data set. So, another optimization idea was to store all different coefficients in an array and put only the integer index to the data in the large geometric array. The advantage is the reduced size of the problem data, as the index needs less memory than the actual coefficient. On the other hand, an additional indirection is introduced by looking up the actual value in the coefficient array every time. Future applications will try to retrieve the conductivity data directly from a CT/MR image. This will result in much more different conductivities, but it might also be possible to discretize these values in a coarse way so that the index idea can still be applied.

Table 7.14 shows the results: The reference is the 1D cluster-based version with the improved data alignment (refer to the code in figure 7.14). A version using the *unsigned char* data type (1 byte) and one using the *int* data type (4 bytes) index variables were tested. These versions allocate a array of *floats* with 255 entries for the coefficients. An additional version (see: *Index unsigned char (fix)* in the table) used an array size of four entries. This is the number of coefficients in the test data set.

	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
Reference	10.33	9.82	4.18	5.12
Index (unsigned char)	10.78	9.52	4.24	5.43
Index (int)	11.56	10.40	5.06	6.62
Index unsigned char (fix)	10.30	10.07	4.28	5.54

Table 7.14: Cycles per iteration using different indexing methods

Table 7.15 shows the cache performance of the different versions on the Pentium Pro. Using a 1 byte index (*char*) reduces the cache misses enormously because of the reduced memory usage. The *int* is as big as a *float*. So it does not save any memory but needs the additional indirection for the data access. Therefore, this version is up to 20% slower on every machine. Using a coefficient array size as small as possible does not have any influence, as expected: The unused entries in the array are never accessed anyway.

```
1     for (curr_cluster=0;curr_cluster<cluster_count;curr_cluster++)
        {
          start=startindex[curr_cluster]+startindex[curr_cluster]%2;
          end=endindex[curr_cluster];
5         for (i=start;i<=end;i+=2)
            {
              inner_loop_count++;
              cc = - (double)( stencils[i-kjsize].xc  + stencils[i].xc +
                               stencils[i-ksize].yc + stencils[i].yc +
10                             stencils[i-1].zc     + stencils[i].zc );

              /* First compute the part of the residual that
                 does not involve the rhs */
              resid =
15              (double)stencils[i].xc            * u[i+kjsize] +
                (double)stencils[i-kjsize].xc     * u[i-kjsize] +
                (double)stencils[i].yc            * u[i+ksize]  +
                (double)stencils[i-ksize].yc      * u[i-ksize]  +
                (double)stencils[i].zc            * u[i+1]      +
20              (double)stencils[i-1].zc          * u[i-1]      +
                cc * u[i];

              /* Now add part that depends on rhs */
              for ( l = 0; l < n_rhs; l++ ){
25              if ( rhs_new[l].coord == i )
                  resid += rhs_new[l].value;
              }

              /* Add scaled correction to value at current node */
30            u[i] -= omega * resid / cc;

              /* Sum up squares of residual */
              resnorm += resid * resid;
            }
35      }
```

Figure 7.14: Red iteration part of the 1D code

All in all, the index idea does not improve the code performance. On all machines (except of the Athlon), there is no significant difference in spite of the enhancement of the cache performance. This leads to the conclusion that accessing the data is not the problem of the current code version. There might be some internal stalls slowing down the processors. A detailed analysis on the Alpha 21164 using DCPI [17] revealed, that the index version (*unsigned char*) spends about 28% (10% less than the reference version) cycles in dynamic stalls (cache misses,...), but about 40% (4% more) in static stalls, mainly because of register A (8.3%) and B (26.7%) dependencies. The percentage of useful cycles (cycles not in stalls) is about 32%. This is not bad considering the memory usage and the data dependencies of the code.

In order to try several improvements mainly against the static stall, the index version is kept because the cache influence of it is much smaller.

7.12 Fine Tuning

In this section several further optimization ideas are tested. The index version with *unsigned char* index variables is used for this due to its good cache behavior. Table 7.16 shows the performance of the different versions. Changes that were kept are marked with a star (*).

At the update of u, the term has to be divided by cc. The first idea was to precalculate $\frac{1}{cc}$. Two different positions were tested for this: Right after the calculation of cc and after the

	L1DCM (thousand)	L2CM (thousand)
Reference	246.5	110.2
Index (unsigned char)	156.2	64.7
Index (int)	239.9	110.8
Index unsigned char (fix)	155.5	65.2

Table 7.15: Cache analysis of the index versions on the Pentium Pro

calculation of the residual. The first version is faster and used for further steps.

Then, a different data arrangement was tested: Putting the three coefficients and the u of a point together. This slows down all processors (nearly 50% on the Pentium Pro) because of a higher memory traffic. So this data arrangement was not used.

Then a two-fold loop unrolling was introduced, meaning that two points of the same color are updated in the body of the inner loop. This is done by doubling every step in the body and applying it to the next point of the corresponding color. The second point has to be checked if it is a head point. This leads to additional conditions in the loop body. As the performance of several loop unrolling versions was worse, this idea was not pursued any longer.

To indicate the possibility of the parallelization of a calculation to the compiler, the calculation of the residual was split in the following way:

```
resid = (double)values[stencils[i].xc]        * u[i+kjsize]  +
        (double)values[stencils[i-kjsize].xc] * u[i-kjsize];
temp2 = (double)values[stencils[i].yc]        * u[i+ksize]   +
        (double)values[stencils[i-ksize].yc]  * u[i-ksize];
temp3 = (double)values[stencils[i].zc]        * u[i+1]       +
        (double)values[stencils[i-1].zc]      * u[i-1]       +
         cc * u[i];
```

The three summands were added right before the update of u. This leads to an improvement on some machines and used for further optimizations. Splitting this calculation in two or six parts is slower. Splitting the calculation of cc in the same manner also slows down the code.

Finally not only $\frac{1}{cc}$ but $\frac{omega}{cc}$ was precalculated. This was also a gain (except on the Athlon) and kept in the code.

	PPRO (millions)	Alpha 1 (millions)	Alpha 2 (millions)	Athlon (user time)
Index (unsigned char)	10.33	9.82	4.18	5.12
Precalc 1/cc (*)	9.77	8.46	3.78	5.46
Coefficients + u	14.60	10.15	4.45	5.55
Loop unrolling	10.41	8.81	3.89	5.62
3-fold resid (*)	9.28	8.32	3.82	5.45
Precalc omega/cc (*)	8.93	8.07	3.68	5.79

Table 7.16: Cycles per iteration using some fine tuning

Figure 7.15 shows the ratio of the cycles per iteration that was achieved by the fine tuning. On the Alpha 21164, that does not support out-of-order execution, the manual tuning results in a gain of approximately 20%. The code does not benefit from the tuning on the Athlon, but the two other processors become about 10% faster.

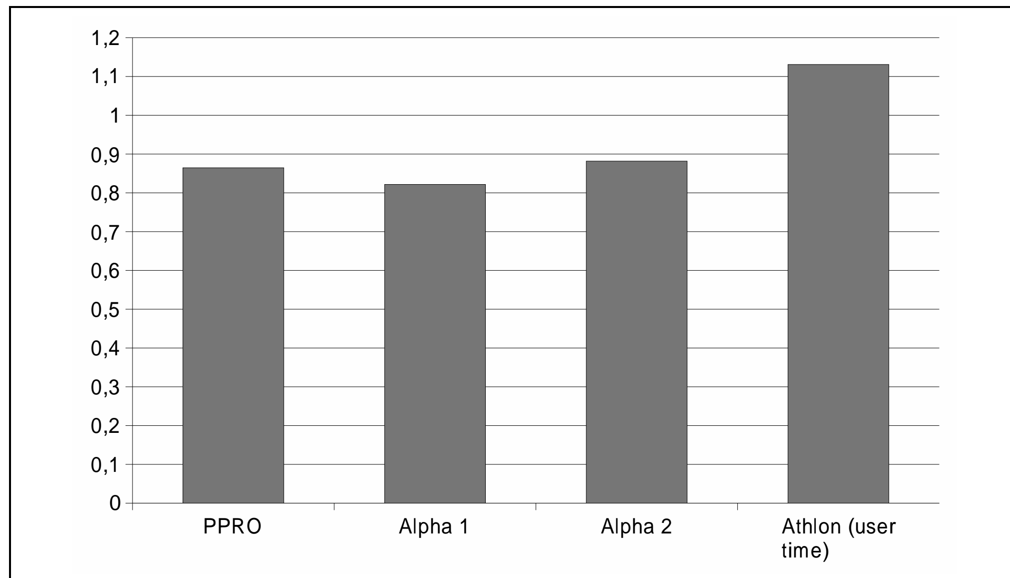


Figure 7.15: Cycles per iteration ratio after fine tuning

This last version was analyzed on the Alpha 21164 with DCPI again. The cycles spent in static stalls (register dependency stalls,...) could be reduced by fine tuning from about 40% to 26%. Now, the dynamic stalls become predominant (from 28% to 38%). All in all, this version has about 36% of useful cycles.

A detailed analysis of the code parts most cycles are spent in shows that about 40% of the cycles are needed for the calculation of cc due to the memory accesses and cache misses (see figure 7.16). Calculating the three residual summands is not that expensive (about 17%). Approximately the same number of cycles is needed at the update of u (9%+6%), mostly due to the dependency on former variables. About the same number of cycles is spent in the RHS check (17%).

```
 4%   for (i=start;i<=end;i+=2)
      {
42%      cc = - (double)(values[stencils[i-kjsize].xc]  + values[stencils[i].xc] +
                         values[stencils[i-ksize].yc]   + values[stencils[i].yc] +
                         values[stencils[i-1].zc]       + values[stencils[i].zc] );

 3%      temp=omega/cc;

 4%      resid = (double)values[stencils[i].xc]             * u[i+kjsize]  +
                 (double)values[stencils[i-kjsize].xc]      * u[i-kjsize];
 3%      temp2 = (double)values[stencils[i].yc]             * u[i+ksize]   +
                 (double)values[stencils[i-ksize].yc]       * u[i-ksize];
10%      temp3 = (double)values[stencils[i].zc]             * u[i+1]       +
                 (double)values[stencils[i-1].zc]           * u[i-1]       +
                 cc * u[i];

17%      for ( l = 0; l < n_rhs; l++ ){
           if ( rhs_new[l].coord == i )
             resid += rhs_new[l].value; }

 6%      resid+=temp2+temp3;
 9%      u[i] -= resid * temp;

 2%      resnorm += resid * resid;
       }
```

Figure 7.16: Cycles distribution in the red iteration part of the final version

Chapter 8

Blocking

This chapter explains a cache optimizing technique which was developed for iterative red/black Gauss-Seidel algorithms. This and some more techniques in this field can be found in [13]. The idea was implemented and tested with the head data set. However, it was not possible to improve the code efficiency with this technique.

8.1 Basic Idea of Blocking

The main idea of blocking techniques in general is to make memory accesses of a code more cache-aware. This means basically optimizing the order of data accesses for data sets larger than the available cache size. [5] explains some basic blocking methods.

See figure 8.1 for a blocking technique for 2D red/black iterations. It shows a grid of red and black points. These are only the points to be updated. So there is an additional row/column on each side of the area that is not updated and not shown here. The number of circles around a point symbolizes the number of iterations already carried out (iteration count). An unoptimized update would work in the following way: First all red points are processed row by row, beginning from the lower left corner here, then all black points.

The blocking technique applied here uses a small *blocking area* for processing, marked by the gray square. Consider the situation in the first picture (a). First, all red points within the square are updated (b). Note, that their black neighbors have all the same iteration count. Then, the square is moved one point to the left and to the bottom (c). In the new square all black points are updated (d). Then, the square is moved right beside the original position (e). This situation is similar to (a) and the whole process continues. So the square is moved through the grid from the left to the right and from the bottom to the top, performing one red and one black update. Whenever the square is partly outside the grid, it has to be clipped to the valid area. The basic advantage of this method is that result data from red updates is needed in the update of the black neighbors. With this method, the biggest part of this is still cached.

This method can be enhanced further: After the red/black update in a square, it can be moved again one row to the left and one row to the bottom and another red/black step can be done and so on, before moving the square to a new place (several levels of iteration).

The same method is applied to a 3D working set using a small cube as an update area instead of the square.

8.2 Results

The 3D blocking technique was applied to the head data set. A modified version of the original code was used to compare the results. This modified version just avoids the right-hand side casts (see section 7.3.1). Several sizes of the blocking area were tested, but no version was significantly better than the reference. Most of them were slower. The best version on the Pentium Pro using an $8 \times 8 \times 8$ blocking cube shows a better level 2 cache behavior than the original version (see table 8.1). The performance of the faster and more significant level 1 cache is worse. An additional version applying several levels of iteration was written, but could not improve the performance as well. So, blocking was not applied in the project. The reasons for this behavior might be the additional overhead due to six nested loops (three for the blocking area's position and three for the update inside the moving cube) and further conditions needed on the margins of the working set. Furthermore, it would cause some additional overhead to adopt this method to the head geometry (i.e. using dynamic loop boundaries, cluster,...), which is important to cope with the already achieved performance gain. The use of padding techniques might improve the performance of the blocking code, but this was not analyzed in the scope of this work.

	Cycles (millions)	L1DCM (thousand)	L2CM (thousand)
Reference	24.97	710.7	266.5
Block size 8	25.22	963.4	170.5

Table 8.1: 3D blocking technique on the Pentium Pro

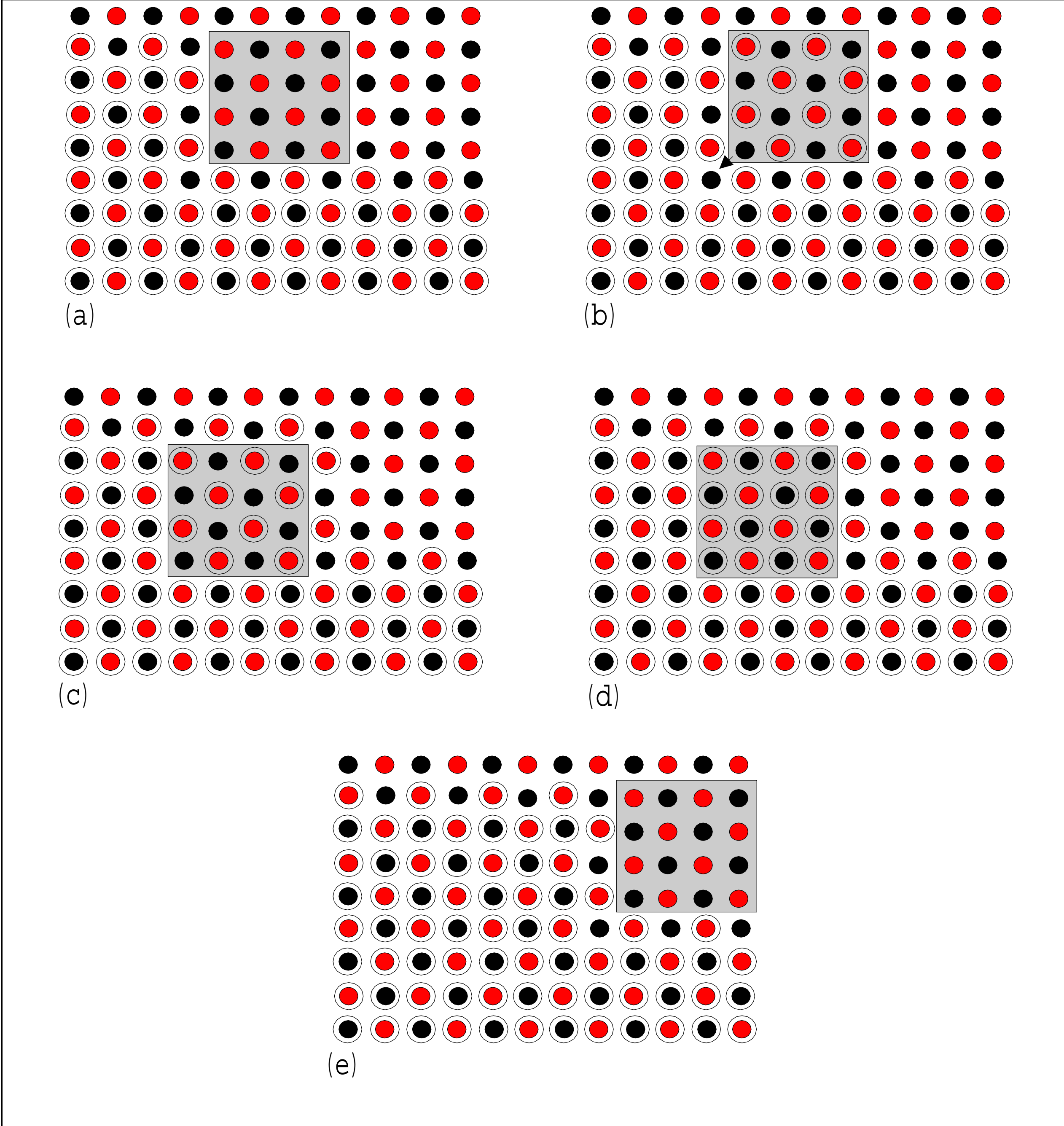


Figure 8.1: Visualization of a 2D blocking method

Chapter 9

Summary

9.1 Overview of the Optimizations

All in all, several different kinds of code optimizations were investigated during this work. The code performance was improved gradually. Therefore, it is not possible to present the influence of different optimization attempts independently here. In order to summarize the optimization ideas applied in the work, table 9.1 was created. It contains most of the code versions of this work in the left column. It shows the results of the modifications in comparison to the corresponding reference code in the work (refer to the optimization chapter for details). So this table should mainly be read in rows, showing which modification leads to which consequence on the different processors.

The table evaluates the cycles per iteration on the processors where PCL is available and the user time on the Athlon. ++ means more than 15% gain in comparison to the reference version and + indicates less than 15% gain. 0 means nearly no change (±3%). − means a loss of up to 15% and −− represents a loss of more than 15%.

	Pentium Pro	Alpha 21164	Alpha 21264	AMD Athlon
cc precalculation	++	++	++	++
Dynamic loop boundaries (y,z)	++	++	++	++
Linear instead of red/black iteration	−			
Avoid RHS cast	++	+	0	+
all_float	−	+	++	++
all_double	−−	−	−−	−
No if in inner loop	−	++	+	0
Cluster-based version (3D)	0	0	+	0
Without RHS check	+	+	+	+
Static data structures	+	−	+	++
All values put together	−	+	−	−−
Three coefficients put together	+	+	++	−−
Precalculate pointer to column	+	+	+	++
RHS check after cc calculation	−−	0	−	−
If only at update of u	0	0	+	0
Cluster-based version (1D)	+	++	++	++
Three stencils (1D)	++	++	+	++
Index (unsigned int)	−	+	0	−
Fine Tuning	+	++	+	−

Table 9.1: Summary table for the optimization steps used in this work

9.2 Most Successful Steps

The most successful steps in optimizing the code were the introduction of dynamic loop boundaries, the avoiding of the right hand side casts, the 1D representation and the array merging (putting all three coefficients of a point together in memory).

Dynamic loop boundaries (clusters in the 1D version) reduce the actual working set of the algorithm to the grid points inside the head. So unnecessary main memory accesses are avoided. The casts of the three coordinates of the right-hand side in the original version lead to unnecessary operations within the inner loop. This slows down the code and was omitted by a new representation. The 1D version was designed to simplify the code by omitting an explicit representation of the 3D nature of the problem. By this, the 3D addressing of points using a threefold pointer dereference could be avoided.

9.3 Overall Gain

A larger head model was used to measure the performance of the optimized code. The new data set has 129 grid points in every dimension. So the memory usage of this set is nearly 8 times higher in comparison to the small data set. Table 9.2 compares the behavior of the original version with the one of the final version.

	Cycles (millions)	L1DCM (millions)	L2CM (millions)
Original Version	277.81	5.58	2.98
Final Version	73.93	1.16	0.80

Table 9.2: Comparison of the original and final code version using a bigger data set

The cycles per iteration on the Pentium Pro is reduced to nearly 25% leading to a speedup of 4. The cache misses are reduced to about 20% in level 1 and 27% in level 2.

Figure 9.1 shows the relative number of cycles per iteration (the relative user time on the Athlon) for the small (65^3 points) and the large (129^3 points) data set.

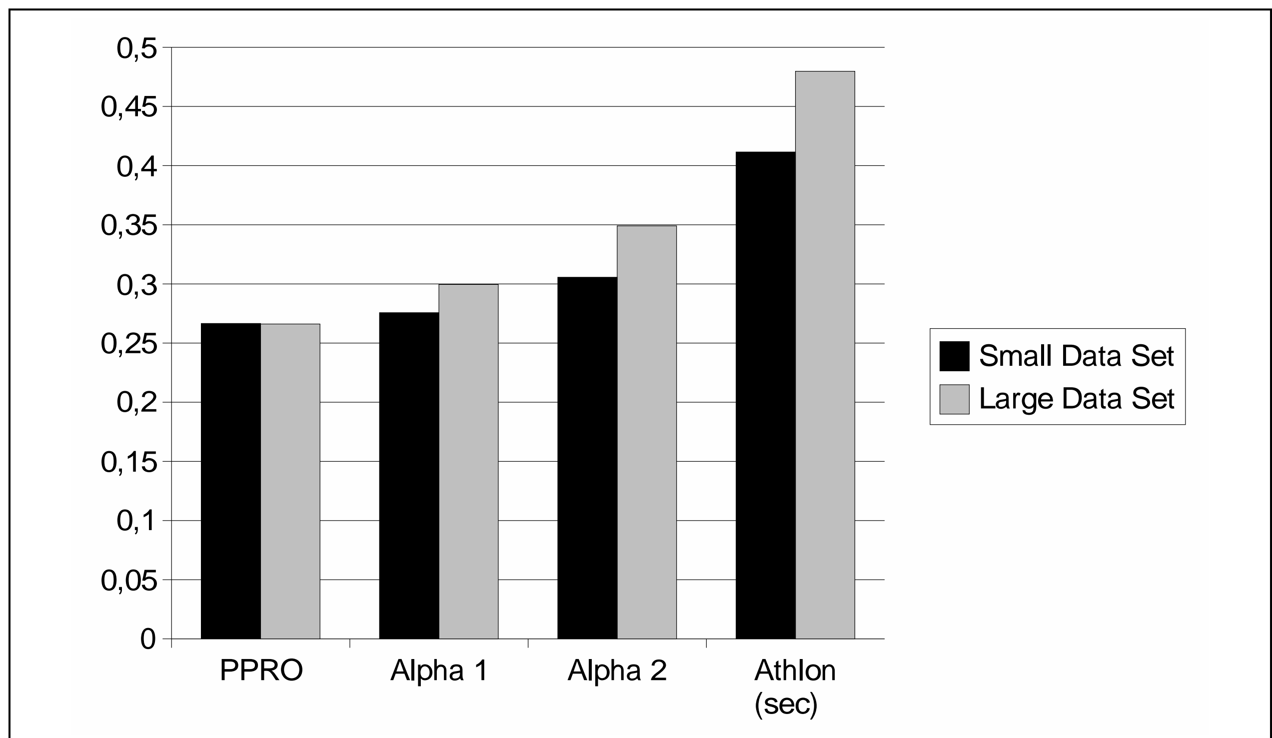


Figure 9.1: Cycles per iteration ratio of the final code compared to the original code

Bibliography

[1] S. Silbernagl, A. Despopoulos, *Taschenatlas der Physiologie*, 4. Edition, Stuttgart, 1991, Thieme Verlag

[2] American Epilepsy Society, *Medical Education Program*, 1999, American Epilepsy Society, www.aesnet.org/educ/medbook.htm

[3] H. R. Schwarz, *Numerische Mathematik*, 4. Edition, Stuttgart, 1997, B.G. Teubner Verlag

[4] A. Berman, R. Plemmons, *Nonnegative Matrices in the Mathematical Sciences (Classics in Applied Mathematics 9)*, December 1994, SIAM

[5] K. Dowd, C. Severance, *High Performance Computing*, 2. Edition, July 1998, O'Reilly & Associates

[6] S. Goedecker, A. Hoisie, *Performance Optimization of Numerically Intensive Codes*, SIAM, to appear

[7] J. L. Hennessy, D. A. Patterson, *Computer Architecture – A Quantitative Approach*, 2. Edition, 1996, Morgan Kaufmann Publishers, Inc.

[8] G. Rivera, C.-W. Tseng, *Data Transformations for Eliminating Conflict Misses*, June 1998, Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal

[9] B. Vanrumste, et al., *The Finite Difference Method and Reciprocity in EEG Dipole Source Localization*

[10] M. Menge, *Superskalare Prozessoren*, Informatik Spektrum, Band 21, Heft 3, pp. 121-130, Juni 1998, Springer Verlag

[11] *AMD Athlon Processor Technical Brief*, December 1999, AMD, www.amd.com

[12] R.E. Kessler, E.J. McLellan, D.A. Webb, *The Alpha 21264 Microprocessor Architecture*, Compaq Computer Corporation, www.compaq.com

[13] C. Weiss, W. Karl, M. Kowarschik, U. Rüde, *Memory Characteristics of Iterative Methods*, Proceedings of the Supercomputing, Conference, Portland, Oregon, 1999, wwwbode.informatik.tu-muenchen.de/Par/arch/cache/

[14] *Pentium Pro Family Developer's Manual Volume 1: Specifications*, 1996, Intel Corporation, www.intel.com

[15] *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 1999, Intel Corporation, www.intel.com

[16] R. Berrendorf, H. Ziegler, *PCL-The Performance Counter Library – A Common Interface to Access Hardware Performance Counters on Microprocessors, Version 1.3*, 1998, Research Centre Juelich GmbH, www.fz-juelich.de/zam/PCL/

[17] *Compaq Continuous Profiling Infrastructure*,

http://www.research.compaq.com/SRC/dcpi/index.html

[18] *Digital Unix Programmer's Guide*, 1996, Digital Equipment Corporation, www.compaq.com

[19] *Dec C Language Reference Manual*, 1997, Digital Equipment Corporation, www.compaq.com

[20] *Digital Unix Assembly Language Programmer's Guide*, 1996, Digital Equipment Corporation, www.compaq.com

[21] *Alpha Architecture Handbook*, 1996, Digital Equipment Corporation, www.compaq.com

Appendix A

Computer Architectures

A.1 Pentium Pro

The Pentium Pro machine used in this work has 96 MB RAM and is running under Suse Linux 5.3. It works at a processor clock rate of 200 MHz.

The processor is three-way superscalar and uses out-of-order execution. It works with the Intel CISC machine instruction set but converts the instructions to up to four so called *uops* internally. For this conversion, three parallel decoders (two for simple instructions (one uop) and one for complex instructions generating up to four uops) are available. So up to six $(4+2)$ uops are generated per cycle. The uops are put into a 40-entry instruction reorder buffer. They are scheduled to one of the five parallel execution units (two integer, two floating point and one memory interface unit), which are 12-stage pipelines. The uops can be executed out-of-order. After the execution three uops can be retired in one cycle.

40 internal registers avoid register dependencies. To the outside (programmer or compiler) eight floating point and eight general-purpose registers are available.

The Pentium Pro supports branch prediction and speculative execution.

The memory system consists of two on-chip 8-Kbyte L1 caches (one four-way-set-associative for instructions and one two-way-set-associative for data) with 32 Byte cache lines. The 256 Kbyte L2 cache is in the same package and supports up to four concurrent accesses. Memory request to the L2 cache or main memory are managed by a memory reorder buffer. The L2 cache is four-way-set-associative and uses also 32 Byte cache lines.

For more information on the Pentium Pro Processor refer to [5], [14] and [15].

A.2 Alpha 21164

This machine uses 128 MB RAM and is running under Digital Unix 4.0 with a clock rate of 600 MHz.

It is a four-way-superscalar processor using a RISC instruction set. It can issue up to four instructions in a cycle but has to wait until all four instructions of a fetch are issued before retrieving the next four instructions. It has a five stage pipeline, where every stage takes one cycle (except of some instructions in the execution stage). It does not support out-of-order execution, but out-of-order completion can occur.

It uses two on-chip L1 8-Kbyte direct-mapped caches with 32 bytes per line. The L1 data cache is write-through and physical. The L1 instruction cache is virtual. The L2 cache is

also on-chip. It is 96 Kbytes, three-way-set-associative and write-back. This machine is fitted with an additional 4 MB direct mapped L3 cache.

More information about the Alpha 21164 can be found in [5].

A.3 Alpha 21264

The Alpha 21264 machine is running under Tru64 Unix with 640 MB RAM. The processor clock rate is 575 MHz.

[12] gives a detailed specification of the features of this processor.

It issues up to four instructions per cycle. These instructions are filled into a 20-entry integer or 15-entry floating point queue. From these four integer and two floating point instructions can be issued in parallel and out-of-order. The integer units can access 80 registers, the floating point units 72 to reduce register dependencies. 64 register are available to the outside using register renaming. Up to 80 instructions can be in progress at once. The Alpha 21264 also supports branch prediction and speculative execution. Furthermore, it supports set-prediction, meaning it tries to predict the best cache set for instructions loaded to the L1 instruction cache.

The memory system receives up to two operations at a cycle. It uses two 64 Kbyte L1 on-chip caches, both two-way-set-associative and 64 Bytes per line. The L1 data cache uses virtual addresses. The L2 cache is 4 MB off-chip, direct-mapped and physically indexed. The system supports eight outstanding loads.

A.4 AMD Athlon

The Athlon is running under Suse Linux Kernel 2.2.16 with 128 MB RAM. The clock rate is 700 MHz.

It also uses the x86 CISC instruction set like the Pentium Pro and translates these instructions to equal-length *MacroOPs*. It can issue three of these MacroOPs per cycle into a 72-entry instruction reorder buffer. From there, these instructions are distributed to the 18-entry integer/address generation scheduler or to the 36-entry floating point scheduler. It features nine independent execution pipelines (three address calculation, three integer and three floating point/MMX/3DNow!).

The L1 cache is divided in 64 Kbytes for instructions and 64 Kbytes for data. Both are two-way-set-associative. The L2 cache size is 512 Kbytes.

More information about this processor can be found in [11].

Appendix B

Source Code of the Final Version

This appendix is a printout of the SOR solver and the preparation step of the final version. It applies 1D red/black iteration using clusters to determine the valid iteration area and indices for improved cache access. The parameters are :

u Result potential vector, initially all set to zero

xc,*yc*,*zc* Coefficients in x-,y- and z-direction

i−,*j*−,*kmax* Maximum index in each dimension

n_rhs Number of right hand side entries in the system of linear equations (usually two)

rhs Coordinates and entries of the right hand sides in the format of the original code, i.e. four doubles per entry

omega Relaxation factor for the SOR iteration

eps Stopping threshold of the iteration (residual norm)

maxit Maximum number of iterations

resnorms Returns the residual norm of every iteration for analysis

```
/* Includes */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "sor3D.h"
#include "matmem.h"

/* Structure for the RHSs */
typedef struct
{
  int coord;
  double value;
} rhs_struct;

/* Structure for the coefficients of a point just storing an integer index to the real value */
typedef struct
{
  unsigned char xc,yc,zc;
} c_struct;
```

```
/* SOR solver function */
int sor3D( double *u, float *xc, float *yc, float *zc, int imax,
           int jmax, int kmax, int n_rhs, double **rhs, double omega,
           double eps, int maxit, double *resnorms )
{
   int i,j,k,l,n;         /* Loop counter                                   */
   int n_sweeps;    /* Counter for performed SOR sweeps               */
   double resid;    /* Residual value at a single node                */
   double resnorm;  /* Euclidean norm of residual                     */
   double cc;       /* Value of central stencil coefficient at a node */
   rhs_struct *rhs_new;  /* RHS after conversion to new representation */
   int isize=imax+1;     /* Number of gridpooints per dimension of the working set */
   int jsize=jmax+1;
   int ksize=kmax+1;
   int kjsize=ksize*jsize;  /* Precalculation of the size of a yz-plain */
   int *startindex,*endindex;  /* cluster variables */
   int curr_cluster;    /* Loop counter */
   int cluster_count;   /* Number of clusters in the dataset */
   int start,end;
   int blank_flag;      /* help variable to find clusters */
   int coord;
   c_struct *stencils;   /* variable for the coefficient data  */
   double temp,temp2,temp3,temp4;
   float values[255];    /* array for index implementation */
   int indexcount=0;     /* number of different coefficients in the data set */

   /* PCL stuff */
#ifdef MACRO_PCL
   int PCLcounters[2];
   PCL_CNT_TYPE int_PCL[2];
   PCL_FP_CNT_TYPE fp_PCL[2];
   int PCLcntrno;
   long perf_result=0;
#endif

   /* Convert rhs to new structure */
   rhs_new=(rhs_struct*)malloc (sizeof(rhs_struct)*n_rhs);
   for ( l = 1; l <= n_rhs; l++ )
   {
      rhs_new[l-1].coord=((int)rhs[l][1])*kjsize+
        ((int)rhs[l][2])*ksize+
         (int)rhs[l][3];
      rhs_new[l-1].value=rhs[l][4];
      printf ("RHS coord %d value %lf\n",rhs_new[l-1].coord,rhs_new[l-1].value);
   }

   /* Prepare coefficients */
   stencils=malloc (sizeof(c_struct)*isize*jsize*ksize);

   /*******************************************************************************************
    *  Preparartion step: one sweep through the data preparing coefficients, index and clusters *
    *******************************************************************************************/
   startindex=NULL;
   endindex=NULL;
   blank_flag=1;
   curr_cluster=-1;
   for (i=0;i<=imax;i++)
     for (j=0;j<=jmax;j++)
       for (k=0;k<=kmax;k++)
       {
          /* 1D coordinate */
          coord=i*kjsize + j*ksize + k;
          /* Stencils and index */
          /* look for this x-coefficient in values */
          for (n=0;n<indexcount;n++)
             if (xc[coord]==values[n])
             {
                stencils[coord].xc=n;
        break;
             }
          if (n==indexcount)   /* not found -> new item */
```

```
   {
      stencils[coord].xc=indexcount;
      values[indexcount]=xc[coord];
      indexcount++;
      if (indexcount==256) fprintf (stderr,"Too many different indexes\n");
   }

   /* look for this y-coefficient in values */
   for (n=0;n<indexcount;n++)
      if (yc[coord]==values[n])
      {
         stencils[coord].yc=n;
         break;
      }
   if (n==indexcount)   /* not found -> new item */
   {
      stencils[coord].yc=indexcount;
      values[indexcount]=yc[coord];
      indexcount++;
      if (indexcount==256) fprintf (stderr,"Too many different indexes\n");
   }

   /* look for this z-coefficient in values */
   for (n=0;n<indexcount;n++)
      if (zc[coord]==values[n])
      {
         stencils[coord].zc=n;
         break;
      }
   if (n==indexcount)   /* not found -> new item */
   {
      stencils[coord].zc=indexcount;
      values[indexcount]=zc[coord];
      indexcount++;
      if (indexcount==256) fprintf (stderr,"Too many different indexes\n");
   }


   /* manage clusters */
   /* points at the margin of the working set may not be updated: set cc=0 here  */
   if (i==0 || i==imax || j==0 || j==jmax || k==kmax || k==0)
      cc=0.0;
   else
       cc = - (double)( xc[coord-kjsize] + xc[coord] +
                        yc[coord-ksize] + yc[coord] +
                        zc[coord-1] + zc[coord] );

   if (cc!=0.0)
   {
      if (blank_flag)
      {
         /* new cluster beginning found */
         curr_cluster++;
         /* allocate new memory if necessary */
         if (curr_cluster%16 == 0)
         {
            startindex=realloc(startindex,(curr_cluster/16+1)*16*sizeof(int));
            endindex=realloc(endindex,(curr_cluster/16+1)*16*sizeof(int));
         }
         startindex[curr_cluster]=coord;
         endindex[curr_cluster]=coord;
         /* switch mode to: look for end-coordinate */
         blank_flag=0;
      }
      else
         /* current coordinate is still in cluster -> adopt end-coordinate */
         endindex[curr_cluster]=coord;
   }
   else
      blank_flag=1;
}
```

```
   cluster_count=curr_cluster+1;
   printf("%d clusters found.\n",cluster_count);

   /********************
    *  Iteration step  *
    ********************/

   n_sweeps = 0;
   resnorm = eps + 1.0;

   while ( n_sweeps <= maxit && resnorm >= eps )
   {
#ifdef MACRO_PCL
      /* Initialize PCL stuff */
      PCLcounters[0]=MACRO_PCL;
      PCLcntrno=1;

      /* PCL Commands to one iteration */
      if (PCLstart(PCLcounters,PCLcntrno,PCL_MODE_USER) !=PCL_SUCCESS)
      {
         fprintf(stderr,"Error in PCLstart2\n");
         exit(-1);
      }
#endif

      /* Increase sweep counter */
      n_sweeps++;

      /* Initialize residual norm to zero */
      resnorm = 0.0;

      /*******************************************************
       *  Sweep over the red points (sum of indices is even)  *
       *******************************************************/

      /* work through all clusters */
      for (curr_cluster=0;curr_cluster<cluster_count;curr_cluster++)
      {
         start=startindex[curr_cluster]+startindex[curr_cluster]%2;
         end=endindex[curr_cluster];

         /* iterate points in the current cluster */
         for (i=start;i<=end;i+=2)
         {
             cc = - (double)(values[stencils[i-kjsize].xc]  + values[stencils[i].xc] +
                             values[stencils[i-ksize].yc]   + values[stencils[i].yc] +
                             values[stencils[i-1].zc]       + values[stencils[i].zc] );
             temp=omega/cc;

             /* First compute the part of the residual that
               does not involve the rhs */
             resid = (double)values[stencils[i].xc]             * u[i+kjsize]  +
               (double)values[stencils[i-kjsize].xc]      * u[i-kjsize];
             temp2 = (double)values[stencils[i].yc]             * u[i+ksize]   +
               (double)values[stencils[i-ksize].yc]       * u[i-ksize];
             temp3 = (double)values[stencils[i].zc]             * u[i+1]       +
               (double)values[stencils[i-1].zc]           * u[i-1]  +
               cc * u[i];

             /* Now add part that depends on rhs */
             for ( l = 0; l < n_rhs; l++ ){
              if ( rhs_new[l].coord == i )
                resid += rhs_new[l].value;
             }

             /* Add scaled correction to value at current node */
             resid+=temp2+temp3;
             u[i] -= resid * temp;

             /* Sum up squares of residual */
```

```
            resnorm += resid * resid;
          }
       }


      /*********************************************************
       *  Sweep over the black points (sum of indices is odd)  *
       *********************************************************/

      /* work through all clusters */
      for (curr_cluster=0;curr_cluster<cluster_count;curr_cluster++)
      {
         start=startindex[curr_cluster]+1-startindex[curr_cluster]%2;
         end=endindex[curr_cluster];
         /* iterate points in the current cluster */
         for (i=start;i<=end;i+=2)
         {
             cc = - (double)(values[stencils[i-kjsize].xc]  + values[stencils[i].xc] +
                          values[stencils[i-ksize].yc]   + values[stencils[i].yc] +
                          values[stencils[i-1].zc]       + values[stencils[i].zc] );
             temp=omega/cc;

             /* First compute the part of the residual that
               does not involve the rhs */
             resid = (double)values[stencils[i].xc]             * u[i+kjsize]  +
               (double)values[stencils[i-kjsize].xc]      * u[i-kjsize];
             temp2 = (double)values[stencils[i].yc]             * u[i+ksize]   +
               (double)values[stencils[i-ksize].yc]       * u[i-ksize];
             temp3 = (double)values[stencils[i].zc]             * u[i+1]       +
               (double)values[stencils[i-1].zc]           * u[i-1]  +
               cc * u[i];

             /* Now add part that depends on rhs */
             for ( l = 0; l < n_rhs; l++ ){
              if ( rhs_new[l].coord == i )
                resid += rhs_new[l].value;
             }

             //if (resid !=0) printf ("Resid %lf\n",resid);
             /* Add scaled correction to value at current node */
             resid+=temp2+temp3;
             u[i] -= resid * temp;

             /* Sum up squares of residual */
             resnorm += resid * resid;
          }
       }



      /*****************************************
       *  Determine Euclidean norm of residual  *
       *  needed for stopping criterion and     *
       *  store it in the history vector        *
       *****************************************/
      resnorm = sqrt( resnorm );
      resnorms[n_sweeps] = resnorm;

#ifdef MACRO_PCL
      /* Stop PCL */
      if (PCLstop(int_PCL,fp_PCL,PCLcntrno) !=PCL_SUCCESS)
      {
         fprintf(stderr,"Error in PCLstop\n");
         exit(-1);
      }
      perf_result+=int_PCL[0];
#endif
    }

   /*************************************************************
    *   Determine why the sweeps stopped. If the residual norm  *
```

```
    *   has become small enough, return the number of sweeps    *
    *   performed, else return -1                               *
    ************************************************************/

   /* cleaning */
   free (startindex);
   free (endindex);
   free (stencils);
   free (rhs_new);

#ifdef MACRO_PCL
   printf("PCL Results(%d): %ld\n",MACRO_PCL,perf_result/n_sweeps);
#endif
   if ( resnorm < eps )
   {
      return n_sweeps;
   }
   else
   {
      return -1;
   }
}
```