Optimizing Locality for ODE Solvers

Thomas Rauber
Institut für Informatik
Universität Halle-Wittenberg
06099 Halle (Saale), Germany
rauber@informatik.uni–halle.de

Gudula Rünger
Fakultät für Informatik
Technische Universität Chemnitz
09107 Chemnitz, Germany
ruenger@informatik.tu–chemnitz.de

ABSTRACT

Runge-Kutta methods are popular methods for the solution of systems of ordinary differential equations and are provided by many scientific libraries. The performance of Runge-Kutta methods does not only depend on the specific application problem to be solved but also on the characteristics of the target machine. For processors with memory hierarchy, the locality of data referencing pattern has a large impact on the efficiency of a program. In this paper, we describe program transformations for Runge-Kutta methods resulting in programs with improved locality behavior. The transformations are based on properties of the solution method but are independent from the specific application problem or the specific target machine, so that the resulting implementation is suitable as library function. We show that the locality improvement leads to performance gains on different target machines. We also demonstrate how the locality of memory references can be further increased by exploiting the dependence structure of the right hand side function of specific ordinary differential equations.

1. INTRODUCTION

The numerical integration of initial value problems (IVPs) of ordinary differential equations (ODEs) plays an important role in the area of scientific computing. Large systems of ODEs arise, e.g., when discretizing time dependent partial differential equations (PDEs) in the spatial domain using the method of lines. This results in an IVP in time with one ODE for each of the spatial discretization points. The advantage of this approach is that for the solution of IVPs, a variable stepsize can be used to adapt the stepsize so that a given accuracy can be guaranteed. Therefore, the number of discretization points in the time domain is usually quite small compared to a global discretization that discretizes the time domain with a fixed stepsize resulting in a (linear or non-linear) equation system.

Runge-Kutta (RK) methods with embedded solutions for stepsize control are one of the most popular methods for the numerical integration of non-stiff IVPs because they combine low execution times with good numerical properties. The idea of these methods is to compute two approximations of different convergence order with the same evaluations of the right hand side function of the ODE system and to use them for stepsize control [10]. Examples are the methods of Fehlberg [8] and Dormand&Prince [13]. Variants of these methods are used in many software libraries like DVERK or RKSUITE [4].

In many scientific applications, not only the arithmetic operations limit the performance on a specific computer but also the movement of data between registers, caches of different levels, main memory, and out-of-core memory. To achieve high performance, programs have to be tuned to make efficient use of the memory hierarchy of the target machine. Tuning can sometimes be performed by a compiler using locality optimizing transformations like loop fusion, blocking, or tiling, see [11, 19] for an overview. Optimizing transformations have been shown to improve the efficiency for matrix computations, algorithms from dense linear algebra, and grid-based computations by reducing the number of cache misses. Cache optimizations are more difficult to achieve for iterative methods that perform several global sweeps over data structures to be computed or updated in every iteration step. For large problem sizes, re-accessing those data in each iteration step can lead to a poor cache exploitation due to capacity misses. For those cases the tuning is often done by hand since subtle rearrangements of the computations are necessary, which are difficult to detect automatically. Cache optimizations for iterative methods in the context of grid-based computations have been investigated, see e.g. [16] and the references therein, but there are only a few approaches for automatic optimizations [12].

In this article, we consider the question how the computations performed by a typical ODE solver can be rearranged such that the locality of the memory references is increased. ODE solver are iterative methods that perform consecutive time steps where the number of steps depends on the step size control mechanism and the time interval in which the ODE has to be integrated. Because of their practical relevance, we consider explicit RK methods with embedded solutions for systems of ODEs. Those methods perform a sequence of consecutive time steps. In each time step a new approximation vector is determined involving the computation of several one-dimensional vectors and the evaluation of function f for different argument vectors, where f is the right hand side function of the ODE system to be integrated. A strategy to improve temporal locality of the memory references is to rearrange these function evaluations such that after the computation of one component of f the result is used as often as possible without performing additional arithmetic operations in between. The goal is to achieve a program structure with a good temporal locality behav-

Proc. of ICS2001, pp. 123-132, June 2001, Sorrento, Italy

ior of the data referencing pattern of the resulting program without destroying spatial locality properties. We make no assumptions about the specific characteristics of the cache memory like cache size, cache line size, cache associativity, or cache replacement strategy. As well, there are no assumptions about the specific application problem, i.e. the specific form of the right hand side function f determining the ODE system to be integrated. Thus, no information about the specific access and dependence structure of f can be used for the transformations to be applied to the general ODE solver. So the difficulties with rearrangements of function evaluations lie in the fact that black-box ODE-solvers are designed to work with arbitrary right hand side functions f so that the specific data dependencies are unknown and, thus, many common loop restructuring methods cannot be applied.

The starting point of our investigation is an RK implementation in the form of routines as they are used in many scientific libraries like RKSUITE. We propose a modification of the original computation scheme which enables rearrangements of the function evaluations without affecting the numerical properties of the method. We demonstrate that the resulting program can be obtained from the original computation scheme by a series of high-level program transformations and restructuring. The modification of the original computation scheme is motivated only by the high-level program structure. So, further evidence is needed to show that the transformations result in more efficient target programs. Runtime tests on different systems demonstrate that the execution time can often be reduced considerably.

The rest of the paper is organized as follows. Section 2 describes the computational structure of explicit RK methods and discusses approaches for increasing the locality. Section 3 describes program transformations for increasing the locality of the memory references. Section 4 presents runtime experiments. Section 5 discusses how the locality can be further improved if the access structure of f is given for the example of the Brusselator equation. Section 6 discusses related work and Section 7 concludes.

2. COMPUTATIONAL STRUCTURE OF RK METHODS

We consider the solution of IVPs of first order ODE systems

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)) \text{ with } \mathbf{y}(x_0) = \mathbf{y}_0 \tag{1}$$

for a given initial vector $\mathbf{y}_0$ at start time x_0 and system size $n \geq 1$. The unknown solution function $\mathbf{y} : I\!R \to I\!R^n$ is approximated numerically. The right hand side function $\mathbf{f} : I\!R \times I\!R^n \to I\!R^n$ is usually a nonlinear function describing the structure of the ODE system.

For non-stiff ODE systems of the form (1), explicit RK methods with an error control and stepsize selection mechanism are robust and efficient [10] and guarantee that the obtained discrete approximation of $\mathbf{y}$ is consistent with a predefined error tolerance [10, 7]. In each time step, these methods compute a discrete approximation vector $\eta_{\kappa+1} \in I\!R^n$ for the solution function $\mathbf{y}(x_{\kappa+1})$ at position $x_{\kappa+1}$ using the previous approximation vector η_κ. To do this, an s-stage RK method computes s stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s \in I\!R^n$ and uses them to compute $\eta_{\kappa+1}$. The following computations are performed:

$$\begin{aligned} \mathbf{v}_1 &= \mathbf{f}(x_\kappa, \eta_\kappa), \\ \mathbf{v}_2 &= \mathbf{f}(x_\kappa + c_2 h_\kappa, \eta_\kappa + h_\kappa a_{21} \mathbf{v}_1), \\ &\vdots \\ \mathbf{v}_s &= \mathbf{f}(x_\kappa + c_s h_\kappa, \eta_\kappa + h_\kappa \sum_{i=1}^{s-1} a_{si} \mathbf{v}_i). \end{aligned} \tag{2}$$

The stage vectors are used to compute the approximation vector $\eta_{\kappa+1}$ and an additional approximation vector $\hat{\eta}_{\kappa+1}$ for error control:

$$\begin{aligned} \eta_{\kappa+1} &= \eta_\kappa + h_\kappa \cdot \sum_{l=1}^{s} b_l \mathbf{v}_l, \\ \hat{\eta}_{\kappa+1} &= \eta_\kappa + h_\kappa \cdot \sum_{l=1}^{s} \hat{b}_l \mathbf{v}_l. \end{aligned} \tag{3}$$

The coefficients $b = (b_1, \ldots, b_s)$, $\hat{b} = (\hat{b}_1, \ldots, \hat{b}_s)$ and $c = (c_1, \ldots, c_s)$ are s–dimensional vectors, and $A = (a_{li})$ is an $s \times s$ matrix specifying the particular RK method under consideration. The order r of approximation $\eta_{\kappa+1}$ and the order $\hat{r}$ of approximation $\hat{\eta}_{\kappa+1}$ usually differ by 1, i.e., $r = \hat{r} + 1$. The difference between the two approximations $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ gives an asymptotic estimate of the local error in the lower order approximation and is used for stepsize control [7]. The approximation of the current step is accepted, if a suitable weighted norm of the local error estimate lies within a predefined tolerance level. Although the estimate of the local error is in the lower order approximation, the more accurate approximation is usually used to advance the integration (local extrapolation). Several embedded RK methods have been derived including the methods of Dormand & Prince (e.g., DOPRI5 of order 5(4) or DOPRI8 of order 8(7)) or Verner's methods DVERK of order 6(5) [10].

The number of function evaluations in one time step of an RK method that are necessary to obtain an approximation of order r increases faster than the order itself. It can be shown that for methods of order 5, 6, or 8, at least 7, 8, or 13 function evaluations, respectively, are necessary [10]. To decrease the runtime of RK methods, parallel implementations have been proposed [15, 14, 5]. In this paper, we investigate runtime improvements for sequential implementations on processors with a memory hierarchy.

Runtime improvements may be achieved by global rearrangement of data accesses for which data dependencies have to be taken into account in order to preserve correctness. The computation scheme (2) and (3) restricts the potential evaluation order due to data dependencies between the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$, and $\eta_{\kappa+1}$ in the following way. All stage vectors have to be computed before the computation of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ starts. Because of the argument vectors of $\mathbf{f}$ in (2), the computation of $\mathbf{v}_i$ depends on $\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}$, so that the stage vectors have to be computed one after another. For a general ODE solver, the dependence structure of $\mathbf{f}$ is not known in advance. We therefore have to make the conservative assumption that every component of $\mathbf{f}$ depends on all vector components of its argument vector, i.e., the computation of one component of $\mathbf{v}_i$ requires that all components of $\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}$ are already determined. The entire dependence structure is illustrated in Figure 1 for the case $s = 4$.

As described, the computation of the argument vector of $\mathbf{f}$ in the calculation of $\mathbf{v}_i$ (see Formula (2)) requires to access all components of $\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}$, so that i vectors of size n have to fit into the cache simultaneously to avoid capacity

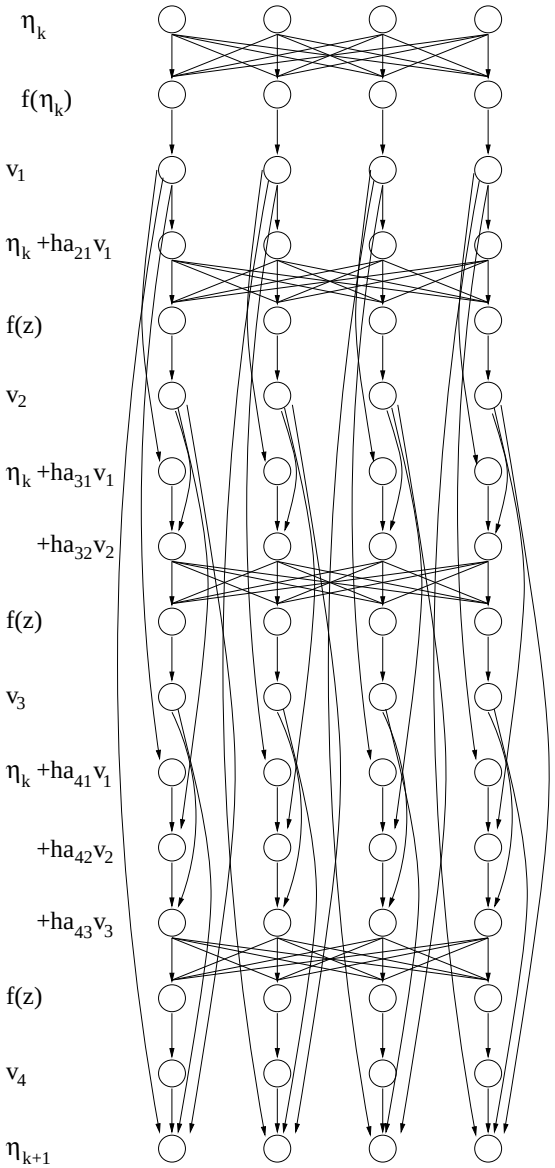

Figure 1: Dependence structure of an explicit RK method with $s = 4$ stage vectors v_1, v_2, v_3, v_4 applied to a system of ODEs with dimension $n = 4$. The circles depict vector elements to be computed. Each horizontal row of circles represents one vector. On the left, the name of the vector in the corresponding row is given. The different argument vectors $\eta_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li}\mathbf{v}_i$, $l = 2, 3, 4$, of f are abbreviated by $\mathbf{z}$. The computation of $\mathbf{z}$ is depicted in the rows above the row of the corresponding $f(\mathbf{z})$. The arrows depict dependencies. The arrows above $f(\mathbf{z})$ show the potential dependency of each component of $f(\mathbf{z})$ on all components of $\mathbf{z}$.

misses. The maximum size $s \cdot n$ of the working set is reached when computing $\mathbf{v}_s$. In the following, we try to reduce the size of the working set and to increase the locality of memory references by suitable program transformations.

3. TRANSFORMATION OF THE RK PROGRAM

In this section, we consider an RK method and transformations to improve the locality properties. We start the transformation process with a standard implementation of the embedded RK method given in computation scheme (2) and (3). Similar implementations are used in scientific libraries. This program version has the loops over the dimension always as innermost loop which has the advantage to provide a good spatial locality since the innermost loop realizes the computations over vectors of length n. We apply a sequence of transformations to this first version with the goal to improve the temporal locality properties without affecting the good spatial locality properties. In order to illustrate the transformation process, we describe some of the intermediate program versions together with the transformations needed to get them. Each intermediate version combines several transformation steps.

3.1 Vector version

The initial version of an embedded RK program can be written in vector notation since the computations of the components of the stage vectors and approximation vectors can be realized by the innermost loop over the dimension $0 \leq j < n$ and since there are no dependencies carried by the innermost loops. In the following core loop of the program the variables in boldface denote entire vectors. The vector $\mathbf{z}$ denotes the argument vector for the function evaluation, and $\mathbf{v}_i, i = 1, \ldots, s$ are the stage vectors; **z1** and **z2** are additional vectors for the computation of the new approximation vector $\eta_{\kappa+1}$. The vector **err** is computed for stepsize control as difference of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. The outermost loop over the time step and the details of the stepsize control are omitted for simplicity. In essence, the body of the iteration over time consists of a loop over the stage vector computation with the computation of the argument vector $\mathbf{z}$ as inner loop. The vector notation has the advantage that the loop structure is simplified and the program transformation process can concentrate on the outer i-loop and l-loop. The body of the time loop is given in the following program fragment. The arrays a, b, and c in the program fragment denote the corresponding coefficients of the RK method. The array bbs denotes the difference of the vectors b and $\hat{b}$.

Program A :

```
(1)   for ( i=0; i<s; i++ ) {
(2)       z = 0.0;
(3)       for ( l=0; l<i; l++ )
(4)           z = z +a[i][l] * v_l;
(5)       z = h * z + η_κ;
(6)       v_i = f( x + c[i] * h , z );
(7)   }
(8)   z1 = 0.0; z2 = 0.0;
(9)   for ( i=0; i<s; i++ ) {
(10)      z1 = bbs[i] * v_i;
(11)      z2 = b[i] * v_i;
(12)  }
(13)  η_{κ+1} = η_κ + h* z2;
(14)  err = h* z1;
```

The program mainly contains non-tightly nested loops. In addition to array computations, the code fragment contains function evaluations of $\mathbf{f}$ where the computation of each component of $\mathbf{f}$ may depend on every component of its argument vector. Therefore, the computation of all components of $\mathbf{z}$ in the program lines (4) and (5) of program **A** is completed before any component of $\mathbf{v}_i$ in line (6) can be computed. The data dependence of the computation of one

component of stage vector $\mathbf{v}_i$ on all components of the stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}$ is realized in the outermost i-loop so that the stage vectors are computed one after another. The computation of the argument and the stage vectors leads to a poor temporal locality especially for large system sizes as each vector component is re-accessed in each loop step after accessing all other components of the vector in between. Improving the temporal locality cannot be achieved by a simple restructuring of the computations. Due to the complex dependencies this requires several intermediate transformation steps.

3.2 Separation of argument vectors

The initial set of transformations modifies the computation of the argument vectors in the program lines (2)–(5) of program **A** to prepare later loop restructurings. At first, separate vectors $\mathbf{z}[i]$ are introduced as argument vectors for different stage vector computations in order to decouple the dependencies and to make further transformations possible. Also, the initialization and the computation of the vectors $\mathbf{z}[i]$ are changed slightly so that the components of the argument vectors are realized within one nested loop. The transformations result in the following program fragment **B** where lines (1)–(6) replace the line (1)–(7) in program **A**.

Program B :

```
(1)   for ( i=0; i<s; i++ ) {
(2)       z[i]= η_κ;
(3)       for ( l=0; l<i; l++ )
(4)           z[i] = z[i] + h* a[i][l] * v_l;
(5)       v_i = f( x + c[i] * h , z[i] );
(6)   }
(7)   z1 = 0.0; z2 = 0.0;
(8)   for ( i=0; i<s; i++ ) {
(9)       z1 = bbs[i] * v_i;
(10)      z2 = b[i] * v_i;
(11)  }
(12)  η_{κ+1} = η_κ + h* z2;
(13)  err = h* z1;
```

Still the entire loop structure mainly consists of non-tightly nested loops. The use of s vectors $\mathbf{z}[i]$, i=1, ..., s, to represent the argument vectors instead of only one vector $\mathbf{z}$ increases the memory requirement but not the working set as $\mathbf{z}[i]$ is used only in one loop iteration i, i=1, ..., s. The further goal is to interchange the i-loop and the l-loop in program fragment **B**.

3.3 Loop interchange

Our aim to use stage vector components as soon after their computation as possible can be reached by interchanging the i-loop and the l-loop in the lines (1)–(6) in program **B**. First, the initialization in program line (2) is separated from the computation loop. Then the function evaluation of line (5) within the outer loop is merged with the inner loop and becomes a new last iteration of the inner l-loop. This results in a tightly nested loop which can be interchanged. The loop structure and their dependencies are shown in Figure 2 where the bottom line corresponds to the computation of $\mathbf{v}_i$, $i = 1, \ldots, s$.

After the loop interchange, the loop structure is again converted into a non-tightly nested loop by extracting the function evaluation from the new inner loop and executing it as first operation in the new outer loop, see program fragment **C'**.

Program C' :

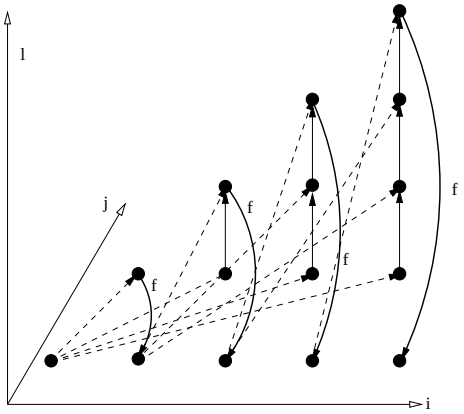

Figure 2: Dependence structure of the i-loop and the l-loop for an explicit RK method with $s = 4$. The circles depict vector components where the entire vector in the j-direction is not shown explicitly. The dependence structure illustrates the dependencies for components of different vectors for one fixed component j. The dependencies between different components due to the evaluation of function f are shown by thick arrows annotated with f. The bottom row circles represent the stage vectors and the new approximation vector. All other circles represent the computation of argument vectors.

```
(1)   for ( i=0; i<s; i++ )
(2)       z[i]= η_κ;
(3)   for ( l=0; l<s; l++ ) {
(4)       v_l = f( x + c[l] * h , z[l] );
(5)       for ( i=l+1; i<s; i++ )
(6)           z[i] = z[i] + h* a[i][l] * v_l;
(7)   }
```

The program lines (3)–(7) show the interchanged loop structure in which a stage vector $\mathbf{v}_l$ is first computed and then immediately used to build all argument vectors for succeeding function evaluations in the same time step. The resulting code has still the drawback of using s stage vectors and s argument vectors, i.e., the code uses more data than the original program. But the new loop structure avoids the interleaving of the accesses to different stage vectors in lines (3)–(7). However, the stage vectors are still used for the computation of **z1** and **z2**. When merging the loop for computing **z1** and **z2** (see program **B**) with the l-loop in line (3) of program **C'**, all interleaved accesses to vectors $\mathbf{v}_l$ are removed and actually only one vector $\mathbf{v}$ is needed to perform the computation of the different stage vectors one after another. The following program results:

Program C :

```
(1)   for ( i=0; i<s; i++ )
(2)       z[i]= η_κ;
(3)   z1 = 0.0; z2 = 0.0;
(4)   for ( l=0; l<s; l++ ) {
(5)       v = f( x + c[l] * h , z[l] );
(6)       z1 = bbs[l] * v;
(7)       z2 = b[l] * v;
(8)       for ( i=l+1; i<s; i++ )
(9)           z[i] = z[i] + h* a[i][l] * v;
(10)  }
(11)  η_{κ+1} = η_κ + h* z2;
(12)  err = h* z1;
```

The resulting program version corresponds to a delay of the function evaluations in computation scheme (2) and (3)

so that a function evaluation is started not before its result is needed for another computation. This is an implementation of the following computation scheme.

Modified computation scheme of an RK method: The computation of argument vectors and stage vectors in the original computation scheme (2) and (3) can be transformed into the computation of modified stage vectors $\mathbf{w}_1$, $\ldots$, $\mathbf{w}_s$ by a delay of the function evaluations using the transformation $\mathbf{v}_i = \mathbf{f}(x_\kappa + c_i h_\kappa, \mathbf{w}_i)$:

$$\begin{aligned} \mathbf{w}_1 &= \eta_\kappa, \\ \mathbf{w}_2 &= \eta_\kappa + h_\kappa a_{21} \mathbf{f}(x_\kappa, \mathbf{w}_1), \\ &\vdots \\ \mathbf{w}_s &= \eta_\kappa + h_\kappa \sum_{i=1}^{s-1} a_{si} \mathbf{f}(x_\kappa + c_i h_\kappa, \mathbf{w}_i). \end{aligned} \tag{4}$$

The approximation vectors $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ are then computed as follows:

$$\begin{aligned} \eta_{\kappa+1} &= \eta_\kappa + h_\kappa \cdot \sum_{l=1}^{s} b_l \mathbf{f}(x_\kappa + c_l h_\kappa, \mathbf{w}_l), \\ \hat{\eta}_{\kappa+1} &= \eta_\kappa + h_\kappa \cdot \sum_{l=1}^{s} \hat{b}_l \mathbf{f}(x_\kappa + c_l h_\kappa, \mathbf{w}_l). \end{aligned} \tag{5}$$

According to the modified scheme a straightforward implementation of the computation scheme requires multiple evaluations of $\mathbf{f}(\mathbf{w}_i)$, $i = 1, \ldots, s-1$. This computational redundancy requires too much execution time, especially when the evaluation of the components of $\mathbf{f}$ is costly, but can be avoided by saving the results of the function evaluations in separate vectors requiring additional vectors for the computation scheme.

After each evaluation of a component of $\mathbf{f}(\mathbf{w}_i)$, the computation scheme (4) also allows the update of the corresponding components of all vectors $\mathbf{w}_j$, $j > i$, and of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. Therefore, no explicit storage of the results of the function evaluations is necessary and temporal locality for the result value and spatial locality for the updated vectors is established. This is realized in program version **C** of the RK method.

3.4 Loop interchange with dimension loop

Further optimizations are possible by a loop interchange with the dimension loop and the use of only one scalar variable to represent all stage vector components. To this end, we make the innermost loops over the dimension explicit. The innermost loops for the vector computation and the loop for updating the argument vectors are interchanged and the resulting dimension loops are combined with the vector loops computing the vectors **z1** and **z2** by loop fusion. The resulting program has no interleaved use of different stage vector components so that the stage vector computations can be represented by a single variable fx. This results in the following program fragment in which vector components are denoted by subscripts.

Program D :

```
(1)   for ( i=0; i<s; i++ )
(2)       z[i]= η_κ;
(3)   z1 = 0.0; z2 = 0.0;
(4)   for ( l=0; l<s; l++ ) {
(5)       for (j=0; j<n ; j++) {
(6)           fx= f_j ( x + c[l] * h , z[l] );
(7)           z1_j = bbs[l] * fx;
(8)           z2_j = b[l] * fx;
(9)           for ( i=l+1; i<s; i++)
(10)              z_j[i] = z_j[i] + h* a[i][l] * fx;
(11)      }
(12)  }
(13)  η_{κ+1} = η_κ + h* z2;
(14)  err = h* z1;
```

The transformation does not only lead to a better temporal locality but also reduces the number of storage locations used within the program which may lead to further reduction of the access distances.

This kind of transformation could not have been applied to the vector version given in program **A** as a complete decoupling of vector computations is not possible for this version. The reason is that in program **A** the computation of the argument vector **z** can be changed in a similar style but that the entire vector **z** is needed in the function evaluation in line (6) of program **A** so that a merge of the dimension loops would lead to an incorrect program.

4. RUNTIME EXPERIMENTS

We have implemented the original RK method and RK program versions according to the program fragments **A**–**D** in order to investigate the resulting runtimes and the runtime improvements. As application problem we use a spatial discretization of the Brusselator equation, a two-dimensional time-dependent partial differential equation describing a reaction-diffusion problem of two chemical substances [10]. A spatial discretization with N discretization points in each dimension leads to an ODE system of size $n = 2N^2$. Each block of N^2 unknowns represent the concentration of one of the chemical substances. As RK method we use the DOPRI5 method with $s = 7$ stages; this is one of the most popular RK methods in practice. The programs are written in C with double precision.

Implementation on Pentium III Figure 3 shows the runtimes of all four program versions of the embedded RK method for different ODE system sizes on a 700 MHz Pentium III processor (having a 4-way associative L1-data cache of size 16 KB and an 8-way associative L2-cache of size 256 KB with line size 32 Byte). The labeling in the Figure has the following correspondence: vector version = Program A; separate argument vector = Program B; loop exchange = Program C; loop exchange cache opt. = Program D. The figure shows that each transformation leads to performance gains on the Pentium III processor. Even the introduction of additional argument vectors improves the runtime as the actual working set in not increased and the increase of the arithmetic operations is compensated by a change in the initialization. The transformation to program **C** results in a further reduction of the runtime solely reached by the loop interchange and the corresponding change of the computation order, i.e., the stage vectors are added to all vectors as soon as they are computed. Finally, the loop interchange with the dimension loop in program **D** and the reduction of the working set by using only one variable for all components of all stage vectors results in another significant reduction of the runtime. The last program version in the figure results from another program modification which we address later in this section.

Figure 4 shows the corresponding MFLOPS rates which are obtained by the PCL library [2] using hardware counters. For small system sizes, the MFLOPS rate first decreases and starting at about $n = 21632$ (corresponding to $N = 104$) remains almost constant. The four different versions **A**–**D** have different MFLOPS rates where the original version

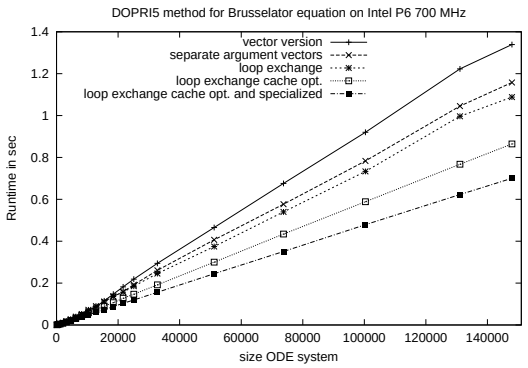

Figure 3: Runtimes of the different RK versions in seconds on Pentium III, 700 MHz.

has the smallest rate, the last version has the highest rate. Versions **B** has a higher rate than version **C**. The change in the MFLOPS rate can be analyzed by considering the cache miss rates.

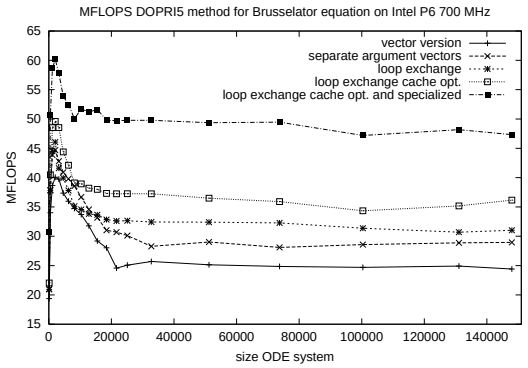

Figure 4: MFLOPS of the different RK versions on Pentium III, 700 MHz.

The L1 cache miss rate in Figure 5 remains almost constant. Only program versions **C** and **D** have a smaller L1 cache miss rate for small system sizes and have a jump at about n=20000. However, the L1 cache miss rate is not responsible for the differences in the MFLOPS rates and the runtimes as three out of the four program versions have almost identical L1 cache miss rates. Only program version **C** has a higher miss rate. Differences in the runtime are caused by L2 cache misses.

Figure 6 shows the L2 cache miss rate, i.e. the number of L2 cache misses per L1 cache miss. The L2 cache miss rates of the four program versions have the same overall behavior. Starting with very small L2 cache miss rates for small system sizes, the rate increases rapidly with increasing system size up to a system size of about $n = 20000$ where a local minimum is reached resulting from a higher L1 cache miss rate. After another slight local minimum at about $n = 50000$ the L2 cache miss rate increases linearly with increasing system size. Considering a fixed system size, the program version **A**–**D** have decreasing L2 cache miss rates with exceptions for small system sizes. This decrease corresponds to the order of the program transformation, i.e. version **A** has the highest values and the values get smaller for the following versions.

The ratio of memory references to floating point operations is given in Figure 7. Again, all four program versions show the same overall behavior. First, this rate decreases up to a local minimum at about $n = 20000$ and after another local minimum at $n = 50000$ the rate increases linearly with the system size where version **A** shows the highest values and version **D** the smallest.

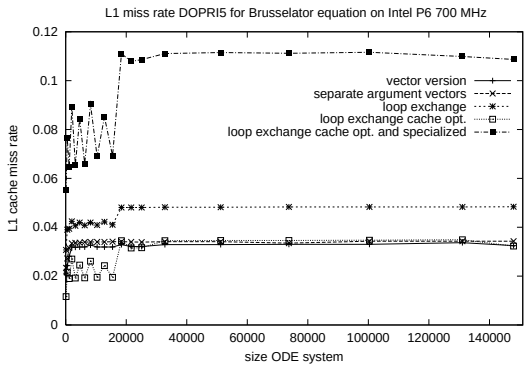

Figure 5: L1 cache miss rate of the different RK versions on Pentium III, 700 MHz.

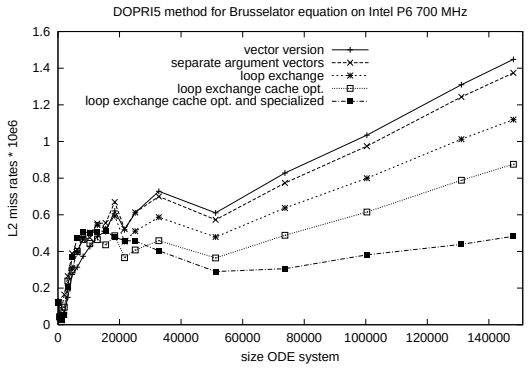

Figure 6: L2 cache miss rate of the different RK versions on 700 MHz Pentium III. Actual values *10e6.

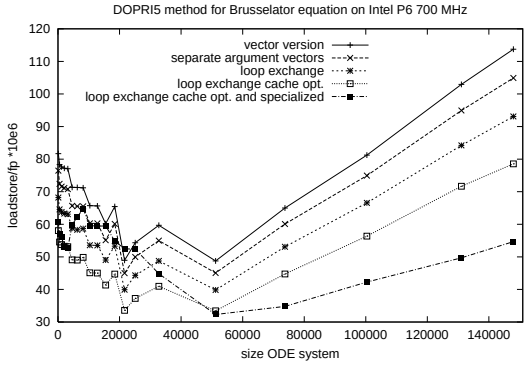

Figure 7: LS/FP rate of the different RK versions on 700 MHz Pentium III. Actual values *10e6.

In order to investigate the effect of the transformations on the runtime separated from other influences, the measurements shown above are performed for RK program versions **A**–**D** in the most general case. This means the implementations make no assumptions about the specific application problem to be solved or the specific RK method to be used.

So, the runtimes and MFLOPS measurements reflect only the effect and improvement caused by the transformations described in Section 3 and do not show secondary effects caused by exploiting special memory reference pattern. As mentioned earlier, the independence from the application problem requires an RK implementation with a general right hand side function f assuming all possible data dependencies on the argument vector. Specific functions may have a more restricted reference pattern which can only be exploited when the RK method is written only for this function. To show the most general case, we have implemented the discrete Brusselator equation in a entirely separate program module realizing the specific access structure in which the integer and comparison operations dominate the floating point operations. The independence from the specific RK method means that no specific RK coefficients b, c, and A are coded but that the RK method contains array references to be linked to an arbitrary RK method.

The general RK-form has the advantage to show only runtime effects caused by the transformations but has the drawback to result in runtimes and MFLOPS rates that are not the best values possible for an RK method solving the Brusselator equation. As an illustration we have included the runtimes for a specific RK implementation of version **D** directly using the values for the RK coefficients b, c, and A of the DOPRI5 method. No adaption to the specific access structure of the Brusselator equation has been made. Figures 3 and 4 show that this optimization improves the resulting runtimes and MFLOPS rates considerably. Similar tuning can be applied to all four versions.

Experiments with blocking. We have also investigated program versions with an additional block structure for the computation of the stage vectors. This program version is a mixture of program **C** and **D**, i.e., starting from program version **C** not the entire vector loop is interchanged with the i-loop but the vector computation is first decomposed into blocks of equal size resulting in a nested loop for the dimension loop and only the outer loop is interchanged with the i-loop. We have tested different block sizes. However, the block version has reached no runtime improvement over the version **C**.

Runtimes on different processors. Figures 8–13 show the runtimes of the four RK implementations on several other processors, a 300 MHz IBM Power PC, a 700 MHz AMD Athlon processor, one 600 MHz DEC Alpha 21164 processor of the Cray T3E, a 300 MHz Sun UltraSparc processor, a 300 MHz MIPS/QED RM5200 processor, and a 195 MHz MIPS R10000 processor. The figures show that the proposed transformations can considerably reduce the runtime on recent processors like the Athlon processor with a similar cache structure as the Pentium III. On other processors the differences are sometimes smaller. Thus, the locality optimizing transformations are suitable on a large range of processors as they increase the locality on processors where it is needed and preserve the performance on processors on which the locality is not so important.

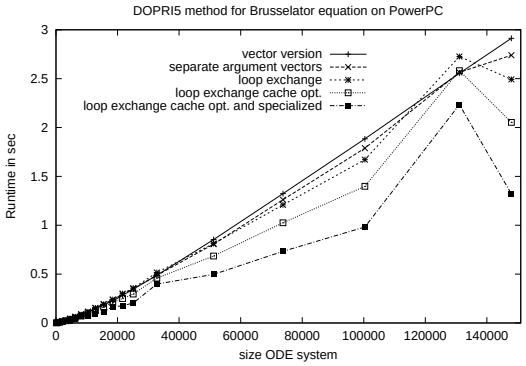

Figure 8: Runtime of the RK program versions in seconds on a 300 MHz Power PC.

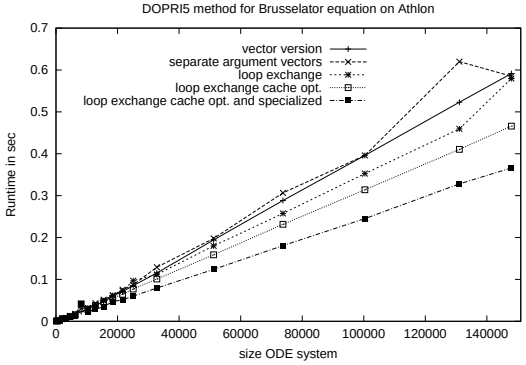

Figure 9: Runtime of the RK program versions in seconds on a 700 MHz Athlon processor.

5. EXPLOITING SPECIFIC ACCESS STRUCTURES

The transformation steps in Section 3 did not assume any specific dependence structure for the right hand side function $\mathbf{f}$ of the ODE system to be integrated. To apply only correctness preserving transformations, we have assumed that any component of $\mathbf{f}$ may access each component of its argument vector. However, many application problems are described by a right hand side function $\mathbf{f} = (f_1, \ldots, f_n)$ with component functions f_l which actually need only a few components of the entire argument vector depending on its index $l = 1, \ldots, n$.

As an example, we consider the discretization of the 2D-Brusselator equation

$$\begin{aligned} \frac{\partial u}{\partial t} &= 1 + u^2 v - 4.4u + \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \\ \frac{\partial v}{\partial t} &= 3.4u - u^2 v + \alpha \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) \end{aligned} \qquad (6)$$

for $0 \le x \le 1$, $0 \le y \le 1$, $t \ge 0$, which describes the reaction of two chemical substances with a diffusion term [10]. The unknown functions u and v describe the concentrations of the two substances. A Neumann boundary condition

$$\frac{\partial u}{\partial n} = 0, \qquad \frac{\partial v}{\partial n} = 0,$$

and the initial conditions

$$u(x, y, 0) = 0.5 + y, \quad v(x, y, 0) = 1 + 5x$$

are used. A standard discretization of the spatial derivatives on a uniform grid with mesh size $1/(N-1)$ leads to the

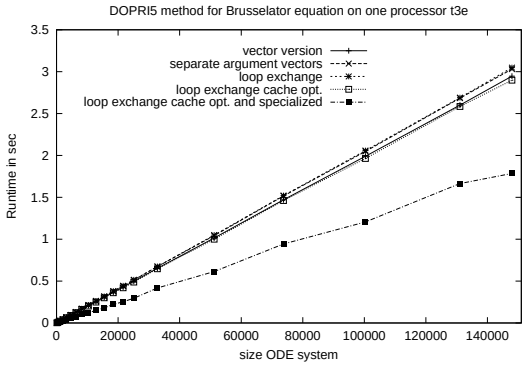

Figure 10: Runtime of the RK program versions in seconds on DEC Alpha processor of the Cray T3E.

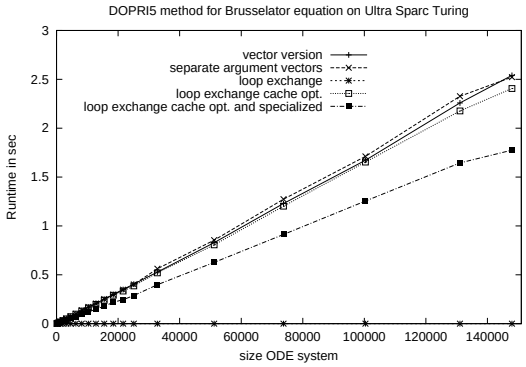

Figure 11: Runtime of the RK program versions in seconds on a 300 MHz UltraSparc.

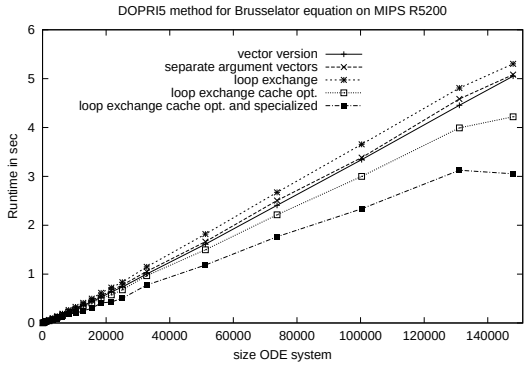

Figure 12: Runtime of the RK program versions in seconds on a 300 MHz QED RM5200.

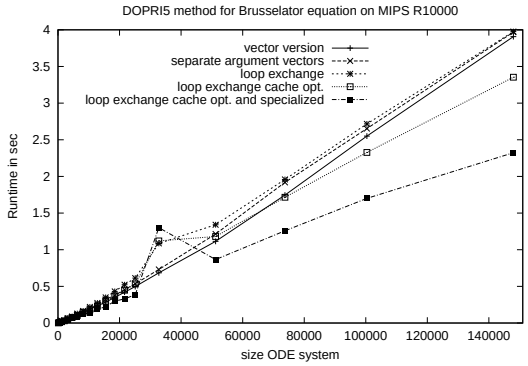

Figure 13: Runtime of the RK program versions in seconds on a 195 MHz MIPS R10000.

following ODE system of dimension $2N^2$:

$$\begin{aligned}\frac{dU_{ij}}{dt} &= 1 + U_{ij}^2 V_{ij} - 4.4U_{ij} + \\ &\quad \alpha(N-1)^2 \left(U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1} - 4U_{i,j}\right) \ , \\ \frac{dV_{ij}}{dt} &= 3.4U_{ij} - U_{ij}^2 V_{ij} + \\ &\quad \alpha(N-1)^2 \left(V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} - 4V_{i,j}\right) \ .\end{aligned}$$

For $\alpha = 2 * 10^{-3}$, this is a non-stiff ODE system, so an explicit RK method is well-suited for the integration. Using a row-oriented assignment of the unknowns U_{ij} and V_{ij}, $i, j = 1, \ldots, N$, onto components of the vector $\mathbf{y}$ or η_κ, respectively, results in the following access structure of the corresponding right hand side function $\mathbf{f}$:

- function component f_l accesses argument components $l-N, l-1, l, l+1, l+N,$ and $l+N^2$ (if available) for $l = 1, \ldots, N^2$,
- function component f_l accesses argument components $l-N, l-1, l, l+1, l+N,$ and $l-N^2$ (if available) for $l = N^2+1, \ldots, 2N^2$.

The access structure of the Brusselator equation is typical for ODE systems resulting from a discretization of a two-dimensional partial differential equation. For the specific case, there is the disadvantage that for the computation of each component f_l a component of the argument vector in distance N^2 is accessed. This access pattern results from the coupling of the original differential equations (6) and can also be observed in the corresponding ODE system. Therefore, no locality improvement can be exploited. An alternative assignment of the unknowns U_{ij} and V_{ij} to the vector $\mathbf{y}$ merges corresponding components of U and V such that component V_{ij} is assigned next to component U_{ij} with the same index values i and j. For this assignment, the vector $\mathbf{y}$ has the entries

$$U_{11}, V_{11}, U_{12}, V_{12}, \ldots, U_{ij}, V_{ij}, \ldots U_{NN}, V_{NN}. \qquad (7)$$

Using this order changes the access structure of the corresponding $\mathbf{f}$ into the following:

- function component f_l accesses argument components $l-2N, l-2, l, l+1, l+2, l+2N$ (if available) for $l = 1, 3, \ldots, 2N^2-1$,
- function component f_l accesses argument components $l-2N, l-2, l-1, l, l+2, l+2N$ (if available) for $l = 2, 4, \ldots, N^2$.

For this access structure the most distant components of the argument vector to be accessed for the computation of one component of $\mathbf{f}$ have distance $2N$. When using RK scheme (4) and (5), the access structure can be exploited in a pipelined computation order for blocks of the stage vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ in the following way: The stage vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ are divided into blocks of size $2N$ each. Using computation scheme (4), the computation of the stage vectors can be initialized by computing s blocks of stage vector $\mathbf{w}_1$. Since the computation of component $(\mathbf{w}_2)_l$ requires the

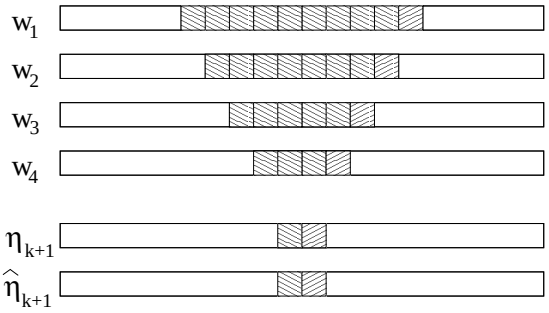

Figure 14: Pipelining computation for 7. Blocks of stage vectors that are accessed for the Brusselator equation for the computation of one block i of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ in the case $s = 4$. Computing block $i + 1$ requires accessing one additional block of each of the stage vectors $\mathbf{w}_1, \ldots, \mathbf{w}_4$.

evaluation of $f_l(x_\kappa, \mathbf{w}_1)$ and since $\mathbf{f}$ has the specific access structure described above, this enables the computation of $s-1$ blocks of $\mathbf{w}_2$, which again enables the computation of $s-2$ blocks of $\mathbf{w}_3$ and so on. This process continues until finally one block of $\mathbf{w}_s$, $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$, respectively, is computed. After the computation of the first block of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$, the next block can be determined by computing only one additional block of $\mathbf{w}_1$ which enables the computation of one additional block of $\mathbf{w}_2, \ldots, \mathbf{w}_s$. This computation is repeated until the last block of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$, respectively, is computed.

The advantage of the pipelining approach is that only those blocks of the stage vectors are kept in the cache that are needed for the further computations of the current time step.

For the computation of an arbitrary block i of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$, the corresponding block i of $\mathbf{w}_1, \ldots, \mathbf{w}_s$ and the neighboring blocks $i-1$ and $i+1$ have to be accessed because of the dependence pattern of $\mathbf{f}$. For the computation of blocks $i-1$ and $i+1$ of $\mathbf{w}_s$, blocks $i-2$ and $i+2$ of $\mathbf{w}_{s-1}$ have to be accessed, too. Fig. 14 shows an illustration. Altogether, at most

$$\sum_{i=1}^{s}(2i+1) = s + s(s+1) = s(s+2)$$

blocks of $\mathbf{w}_1, \ldots, \mathbf{w}_s$ of size $2N$ have to be accessed and should therefore be held in cache simultaneously. Additionally, the two blocks of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ that are just being computed have also to be held in cache. Thus, for the DOPRI5 method with $s = 7$ stages, at most 65 blocks would have to be kept in cache to minimize the number of cache misses. Depending on the grid size N, this is only a small part on the N blocks of size $2N$ that each stage vector contains. Taking $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ into consideration, the proportion of the total number of blocks that have to be held in cache is

$$\frac{s(s+2)+2}{(s+2)N} = \frac{s}{N} + \frac{2}{(s+2)N}$$

with usually $s << N$. For the computation of the first s blocks and the last s blocks of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$, less than $s(s+2)$ blocks of $\mathbf{w}_1, \ldots, \mathbf{w}_s$ have to be accessed because the corresponding values are at the grid boundaries. After the computation of block i of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$, the computation of block $i+1$ accesses many blocks of $\mathbf{w}_1, \ldots, \mathbf{w}_s$ that have already been accessed for the computation of block i. Only one block of each of the stage vectors will be newly accessed. Correspondingly, one block of each stage vector that has been accessed for the computation of block i of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ will not be accessed for the computation of block $i+1$.

This example shows that the access structure and also the storage scheme used might affect the locality behavior for a specific ODE system resulting from, e.g., the discretization of a specific PDE. The knowledge of the specific access structure can be used to structure the code of the ODE method such that a better locality can be obtained than for the general case. If such an application-specific restructuring is used, the resulting ODE solver can only be used for this application. Applying this specific ODE solver to an ODE system that does not exhibit the required access behavior of f may lead to incorrect code if f accesses more elements of its argument vector than expected.

6. RELATED WORK

Because of their large impact on the performance, optimizations to increase the locality of memory references have been applied to many methods from numerical linear algebra including factorization methods like LU, QR and Cholesky [6] and iterative methods like 2D Jacobi [9] and multi-grid methods [16]. Many popular scientific libraries like LAPACK [1] are based on the BLAS (Basic Linear Algebra Subprograms), which can be considered as a de facto standard for the formulation of vector and matrix based numerical algorithms. The BLAS themselves are just a specification of the syntax and semantics of the operations, but many computer vendors provide efficient implementations of the BLAS for specific machines, in particular making effective use of the memory hierarchy of the machine.

Based on BLAS, there are efforts like PHiPAC (Portable High Performance ANSI C) [3] and ATLAS (Automatically Tuned Linear Algebra Software) [18] to provide efficient implementations of BLAS routines. ATLAS for example aims at the automatic generation of efficient BLAS routines by providing a code generator for the automatic creation of optimized on-chip (L1 cache) BLAS operations for specific platforms. The code generator determines the optimal blocking and loop unrolling factors by timings on the specific architecture. BLAS operations for larger arrays or matrices are built up from the fixed-size on-chip operations by architecture-independent code which partitions the matrix or vector operands into blocks of the given fixed size and arranges the computations such that L2 cache usage is optimized. PHiPAC takes a similar approach as ATLAS, but instead of forcing all problems to an independently optimized fixed format, PHiPAC directly optimizes each individual operation.

As mentioned earlier, there are also approaches for other dense linear algebra algorithms or grid based methods [6, 9] but not for ODE solvers. The cache performance of two- and three-dimensional multigrid algorithms is investigated in [17]. In particular, a 2D red-black Gauss-Seidel relaxation method as the most time-consuming part of a multigrid method is considered and a blocking technique and array padding are applied to reorder the data accesses for increasing the temporal locality. The increased data locality leads to significantly larger MFLOPS rates especially for grids with a fine discretization and many grid points.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have investigated the locality behavior of

embedded RK methods. Embedded RK methods are popular ODE-solvers and exhibit the typical iterative structure of one-step methods for the solution of initial value problems. We have presented program transformations motivated by the goal to improve the temporal locality in each time step without affecting the good spatial locality of standard RK implementations. The resulting program has been generated from the standard method by a sequence of correctness preserving transformation steps. The transformation steps were guided by the aim to improve the temporal locality of the stage vector computations which requires several intermediate transformations to make other transformations possible. We have shown that the optimized program shows better efficiency on several processors. It can be observed that the locality improvements have a larger effect for recent processors, like the Pentium III processor or the Athlon processor.

Future work will include the investigation of similar transformation steps for other important ODE solvers, like multistep solvers or implicit solvers for stiff ODEs. The goal will be to improve the performance of standard library implementation but also to improve special purpose solvers of specific application programs. One example is the Brusselator equation mentioned in this paper. The transformation process did not improve the performance in each step but several intermediate transformations were needed to make an improving optimization possible. So, another question is which parts of transformation processes as described in this paper can be automated and what measure can be used to guide the transformations.

Acknowledgement

We thank the NIC Jülich for providing access to the Cray T3E and making the PCL library available.

8. REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarlin, A. McKenney, and D. Sorensen. *LAPACK Users' Guide, Third Edition*. SIAM, 1999.

[2] R. Berrendorf and B. Mohr. *PCL - The Performance Counter Library, Version 2.0*. Research Centre Juelich, September 2000.

[3] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *11th ACM Int. Conf. on Supercomputing*, 1997.

[4] R.W. Brankin, I. Gladwell, and L.F. Shampine. *RKSUITE release 1.0*, 1991.

[5] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford Science Publications, 1995.

[6] J. Choi, J.J. Dongarra, L.S. Ostrouchov, A.P. Petitet, D.W. Walker, and R.C. Whaley. Design and Implementation of the ScaLAPACK LU, QR and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996.

[7] Wayne H. Enright, Desmond J. Higham, Brynjulf Owren, and Philip W. Sharp. A Survey of the Explicit Runge-Kutta Method. Technical Report 94-291, University of Toronto, Department of Computer Science, 1995.

[8] E. Fehlberg. Classical fifth–, sixth–, seventh– and eighth order Runge–Kutta formulas with step size control. *Computing*, 4:93–106, 1969.

[9] K.S. Gatlin and L. Carter. Architecture-Cognizant Divide and Conquer Algorithms. In *Proc. of Supercomputing'99 Conference*, 1999.

[10] E. Hairer, S.P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer–Verlag, Berlin, 1993.

[11] François Irigoin and Rémi Triolet. Supernode partitioning. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, Calif., January 1988.

[12] M. Kandemira, J. Ramanujam, and A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed memory machines. *Journal of Parallel and Distributed Computing*, 60:924–965, August 2000.

[13] P.J. Prince and J.R. Dormand. High order embedded Runge-Kutta formulae. *J. Comp. Appl. Math.*, 7(1):67–75, 1981.

[14] T. Rauber and G. Rünger. Diagonal-Implicitly Iterated Runge-Kutta Methods on Distributed Memory Machines. *Int. Journal of High Speed Computing*, 10(2):185–207, 1999.

[15] T. Rauber and G. Rünger. Parallel Execution of Embedded and Iterated Runge–Kutta Methods. *Concurrency: Practice and Experience*, 11(7):367–385, 1999.

[16] L. Stals and U. Rüde. Data Local Iterative Methods for the Efficient Solution of Partial Differential Equations. In *Proc. of Computational Techniques and Applications*, 1997.

[17] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proceedings of the ACM/IEEE SC99 Conference*, Portland, Oregon, November 1999.

[18] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. Technical report, University of Tennessee, 1999.

[19] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.