Scientific Programming 12 (2004) 133–154
IOS Press

Improving locality for ODE solvers by program transformations

Thomas Rauber[a] and Gudula Rünger[b]
[a]*Fachgruppe für Informatik, Universität Bayreuth, 95440 Bayreuth, Germany*
E-mail: rauber@uni-bayreuth.de
[b]*Fakultät für Informatik, Technische Universität Chemnitz, 09107 Chemnitz, Germany*
E-mail: ruenger@informatik.tu-chemnitz.de

Abstract. Runge-Kutta methods are popular methods for the solution of ordinary differential equations and implementations are provided by many scientific libraries. The performance of Runge-Kutta methods depends on the specific application problem to be solved, but also on the characteristics of the target machine. For processors with a 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 implementations with improved locality behavior for systems of ODEs. 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 recent microprocessors.

1. Introduction

The numerical integration of initial value problems (IVPs) of ordinary differential equations (ODEs) plays an important role in many areas of scientific computing. In particular, IVPs can be used to model time-dependent processes, like the simulation of particles under the influence of gravitational or electrical forces, the kinetics of chemical reactions, or dynamical systems describing the evolution of complex systems, see e.g. [11] for a detailed description. Large systems of ODEs also arise when discretizing time dependent partial differential equations (PDEs) in the spatial domain using the method of lines. The result is a large system of ODEs in time with one ODE for each of the spatial discretization points. This semi-discretized approach has the advantage that a variable stepsize can be used for the solution of IVPs to adapt the stepsize and thus to guarantee a given accuracy. Therefore, the number of discretization points in the time domain is usually quite small compared to a global discretization that uses a fixed step size for the time domain resulting in a (linear or non-linear) equation system.

Depending on the characteristics of the IVP resulting from the spatial discretization, explicit or implicit solution methods are used for the integration. Runge-Kutta (RK) methods with embedded solutions for stepsize control are one of the most popular explicit methods for the numerical integration of non-stiff IVPs, since they combine low execution times with good numerical properties. The idea of embedded methods is to compute two approximations for each time step and to exploit these for error control and stepsize selection [15]. The approximations have different convergence order but use the same evaluations of the right hand side function of the ODE system. Examples for embedded methods are the methods of Fehlberg [13] and Dormand and Prince [19]. Variants of these methods are used in many software libraries like RKSUITE [5].

In many scientific applications, not only the number of arithmetic operations and function evaluations influences 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 per-

ISSN 1058-9244/04/$17.00

formed by a compiler using loop transformations like loop fusion [9], loop interchange, or loop tiling [16], see [1,18,27] for an overview. Additionally, data layout transformations like array padding [23], array merging, or array transposition may have to be used to improve the effectiveness of loop transformations by avoiding conflict misses, see [18] and the references therein for an overview. An approach for the automatic application of loop transformations is presented in [17]. 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 each iteration step. Re-accessing those data in each iteration step can lead to poor cache exploitation due to capacity misses. In 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 have been investigated in the context of grid-based computations, see e.g. [24] and the references therein.

ODE solvers are iterative methods that perform consecutive time steps where the number of steps depends on the step size control mechanism, the desired accuracy of the approximation, 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. RK methods are one-step methods and perform a sequence of consecutive time steps. In each time step a new approximation vector is determined by computing several stage vectors and by applying the right-hand side function $\mathbf{f}$ of the ODE system to different argument vectors that are formed by combining the stage vectors in partial sums. To decrease the runtime of RK methods, parallel implementations have been proposed [6,21,22]. Combining different sources of parallelism can be used to obtain scalable parallel implementations. Another key factor in obtaining high sustained performance on a parallel machine is to use an efficient sequential method on each processor that exploits the local memory hierarchy as effective as possible. A parallel realization based on an improved sequential implementation instead of a standard sequential implementation can significantly reduce the overall execution time.

In this paper, we address this issue and investigate how a good per processor performance can be reached by improving the cache utilization. We consider the rearrangment of the computations performed by a typical RK solver such that the locality of memory references is increased. This is particularly important for systems of ODEs that result from the spatial discreatization of PDEs, since the resulting ODE systems are usually quite large and require large computation time. A consecutive computation of the stage vectors may lead to capacity misses if the stage vectors do not fit into the cache. The strategy to avoid capacity misses and to improve temporal locality of memory references is to rearrange the function evaluations in each time step so that the result of the function evaluation of one component of $\mathbf{f}$ is used as often as possible immediately after its computation without performing additional arithmetic operations in between. The goal is to achieve a program structure with good temporal locality behavior of the data referencing pattern without destroying good spatial locality properties. We are interested in performance improvements for a variety of recent microprocessors and we make no assumptions about the specific characteristics of the cache memory, like cache size, cache line size, cache associativity, or cache replacement strategy. The optimizing transformations are guided only by the program structure of the numerical method and its inherent data access pattern. Also we make no assumptions about the specific application problem and so the specific form of the right hand side function $\mathbf{f}$ determining the ODE system to be integrated. This is a typical situation when using an RK method as black-box solver, e.g. in a scientific library implementation. Thus, no information about the specific access and dependence structure of $\mathbf{f}$ can be exploited for the transformations to be applied to a general RK solver.

The starting point of our investigations is an RK implementation in a form used in many scientific libraries. We propose a modification of this original computation scheme which enables further rearrangements of the function evaluations and demonstrate how the resulting implementation can be obtained from the original computation scheme by a series of high-level program transformations. The modifications of the original computation scheme are accompanied by a program analysis that considers the data set size, the working set size, and the sequence of access times to each array element. Runtime tests on different microprocessors demonstrate that the execution times of RK solvers are often reduced considerably. The resulting computation scheme also provides a good basis for a parallel implementation that is suitable for obtaining high sustained performance.

The rest of the paper is organized as follows. Section 2 describes the computational structure of embedded RK methods. Section 3 presents program analysis techniques to analyze the locality behavior. Section 4 describes a series of program transformations for increasing the locality of the memory references of embedded RK methods. Section 5 presents runtime experiments. Section 6 discusses related work and Section 7 concludes.

2. Computational structure of embedded RK methods

We consider the solution of IVPs of first order ODE systems of size $n \geqslant 1$:

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

with a given initial vector $\mathbf{y}_0 \in \mathbb{R}^n$ at start time x_0. The right hand side function $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ describes the structure of the ODE system and can be nonlinear.

For the solution of non-stiff ODE systems of the form Eq.(1), explicit RK methods with an error control and stepsize selection mechanism are robust and efficient [15] and guarantee that the obtained discrete approximation of $\mathbf{y}$ is consistent with a predefined error tolerance [12,15]. In each time step, an explicit RK method computes a discrete approximation vector $\eta_{\kappa+1} \in \mathbb{R}^n$ for the unknown 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 $\mathbf{v}_1, \ldots, \mathbf{v}_s \in \mathbb{R}^n$ according to:

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

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

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

The s–dimensional vectors $b = (b_1, \ldots, b_s)$, $\hat{b} = (\hat{b}_1, \ldots, \hat{b}_s)$ and $c = (c_1, \ldots, c_s)$ and the $s \times s$ matrix $A = (a_{il})$ specify 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 [12]. 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 taken to advance the integration (local extrapolation). Several embedded RK methods have been derived including the methods of Dormand and Prince (e.g. DOPRI5 of order 5(4) or DOPRI8 of order 8(7)) and Verner's methods DVERK of order 6(5) [15]. The number of function evaluations performed in one time step of an RK method 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 [15].

Performance improvements may be achieved by global rearrangements of data accesses, but data dependencies have to be taken into account in order to preserve correctness. The computation scheme Eqs (2) and (3) restricts the potential evaluation order due to data dependencies between the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$, and $\eta_{\kappa+1}$, since all stage vectors have to be computed before the computation of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ can start. Also, according to the form of the argument vectors of $\mathbf{f}$ in Eq. (2), the computation of stage vector $\mathbf{v}_i$ depends on $\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}$, $i = 2, \ldots, s$, 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 and therefore the conservative assumption that every component of $\mathbf{f}$ depends on all vector components of its argument vector has to be made. Hence, the computation of one component of $\mathbf{v}_i$ requires that all components of $\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}$ have already been computed. Since $\mathbf{f}$ may access all these components, i vectors of size n have to fit into the cache simultaneously to avoid capacity misses. The maximum size $s \cdot n$ of data required within one time step is reached when computing $\mathbf{v}_s$. The dependence structure of one time step is illustrated in Fig. 1 for the case $s = 4$.

3. Methods of program analysis for locality improvement

To estimate the effects of program transformations on the execution behavior of the resulting program ver-

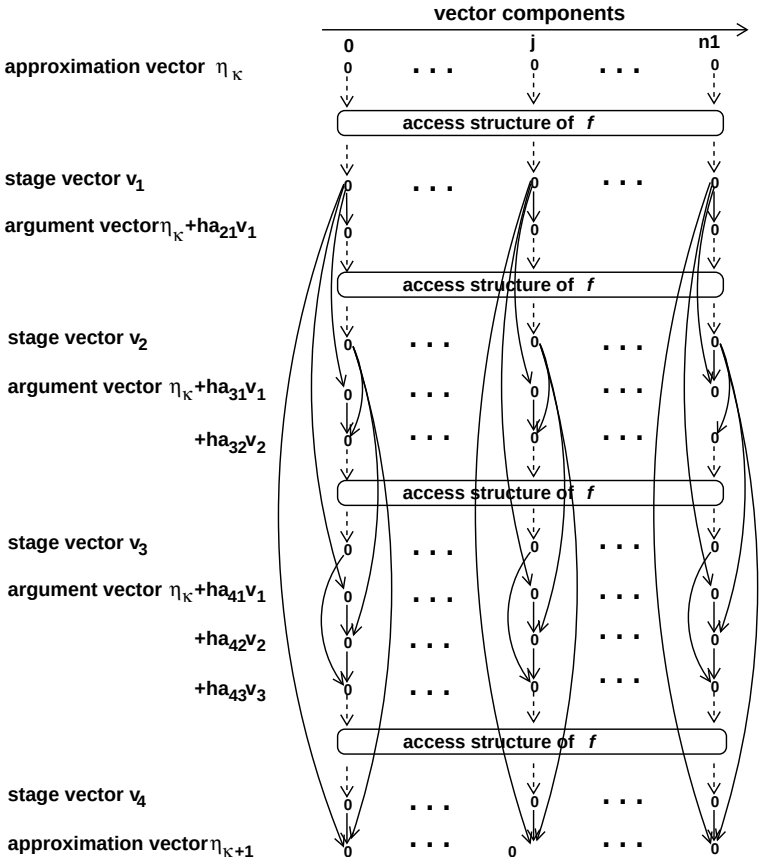

Fig. 1. Dependence structure within one time step of an explicit embedded RK method with $s = 4$ stage vectors computing approximation vector $\eta_{\kappa+1}$ from the previous approximation vector η_κ. Approximation vectors, stage vectors and argument vectors are shown horizontally. Arrows depict dependencies according to the computation scheme Eqs (2) and (3). The access structure of **f** is shown as black-box and can vary according to the specific ODE system to be solved.

sions on cache architectures, we introduce several measures that are used in the following sections. The goal is to capture program properties that influence or determine the locality behavior of the resulting program versions. Although we make general assumptions about the memory hierarchy and consider black-box ODE solvers, it is useful to consider the inherent influence of the system size on the locality properties and the resulting runtime. The connections between system size and locality properties are complex but can be captured by a program analysis.

The *data set* $ds(P)$ of a program version P is the collection of variables that P allocates during its execution. The *data set size* $dss(P)$ of a program version P is the amount of main memory storage that is needed to store the complete data set $ds(P)$. The inherent *working set size* $\mathbf{wss}(P)$ of a program version P is the maximum amount of data that needs to be stored in a fully-associative cache memory at a specific point of program execution. This notion is based on the assumption that a program has a set of data that it reuses substantially for a period of time before moving to other data. The shift between one set of data and another may be abrupt or gradual [8,10]. Using a cache with a smaller associativity may require a cache that is larger than the inherent working set size because of conflict misses. For many programs, the inherent working set size is smaller than the data set size, since some of the variables are used only for specific parts of the program. This is also the case for the program versions of the RK methods considered in the following section.

The locality of memory references may have a large influence on the execution time of a program. It is useful to distinguish between temporal and spatial locality. A program exhibits temporal locality if it tends to access the same memory locations repeatedly in a short time interval of program execution. Spatial locality oc-

curs if the reference to a memory location x takes place shortly before a reference of a memory location y that is close to x in the address space. Spatial locality is formulated at the granularity of individual words and can also be viewed as temporal locality at the granularity of cache blocks or larger memory units. The locality of memory references can be increased by program transformations that do not violate the data dependencies of the program. In Section 4, we consider program transformations to increase the temporal locality of RK methods without destroying spatial locality properties.

The temporal locality of a program version can be described by the access points to the individual storage locations during program execution. To quantify the temporal locality behavior of a program version P, we enumerate the (read and write) memory accesses of P and consider the sequence of access times $S_a(x) = (n_0(x), n_1(x), \ldots, n_{l_x}(x))$ for each storage location x used by P. The temporal locality for an individual storage location x is determined by the sequence of distances $S_d(x) = (n_1(x) - n_0(x), \ldots, n_{l_x}(x) - n_{l_x-1}(x))$ between neighboring access points to the same location x. If the access distances are small, the likelihood that a reuse in the cache occurs is large. The goal of program transformations for improving the temporal locality is to reduce the access distances of some of the storage locations without affecting the access distances of other storage locations. Based on these observations, we define that a program version P_1 has a higher temporal locality than a program version P_2, if P_1 and P_2 use the same working set and if P_1 has smaller access distances than P_2 for at least one storage location x and P_1 and P_2 have the same access distances for all other storage locations $y \neq x$.

Although there may be a complicated relationship between the memory references of a program and characteristics of a cache architecture, like replacement policy, size of the cache blocks and associativity, it can usually be observed that a reduction in the access distances according to the definition above leads to fewer cache misses and thus to a smaller execution time. The effect depends on the working set size and the data set size which is influenced by the system size in the case of ODE systems.

4. Locality improving transformations of the RK program

Starting with a standard implementation of embedded RK methods given by Eqs (2) and (3), we apply a sequence of program transformations with the goal to improve the temporal locality properties without affecting the good spatial locality properties and the dependence constraints. In order to illustrate the transformation process, we describe some of the intermediate program versions together with the transformations needed to obtain them.

4.1. Vector version

The standard implementation of the embedded RK method Eqs (2) and (3) has the loops over the vector 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. The initial version can therefore be written in vector notation. Program A shows the core loop of the program in which the variables in boldface denote entire vectors, see Fig. 2 (left). In addition to the notation introduced in Eqs (2) and (3), the vector $\mathbf{z}$ denotes the argument vector for the distinct function evaluations; $\mathbf{z1}$ and $\mathbf{z2}$ are additional vectors used for the computation of $\eta_{\kappa+1}$; the vector **err** is the difference of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ and is used for stepsize control. The outermost loop over time and the details of the stepsize control are omitted for simplicity. The array *bbs* is the difference of b and $\hat{b}$. Fig. 2 (right) shows the computation order in the iteration space (with the indices starting at 0 to be conform with the program).

Program A mainly contains non-tightly nested loops. The program version meets the dependence contraints shown in Fig. 3 by realizing a strictly consecutive computation order of the stage vectors. The data and working set size of program A are

$$dss(A) = (s+6) \cdot n \text{ and } wss(A) = (s+2) \cdot n$$

since $s+6$ vectors of size n are used but the i-loop in program lines (1)–(7) accesses only $i+2$ vectors for $i = 0, \ldots s-1$, and the computation of the vectors $\mathbf{z1}$ and $\mathbf{z2}$ in lines (9)–(12) separately accesses $s+2$ vectors.

The temporal locality is characterized by the computation of the stage vectors $\mathbf{v}_i$: The vectors are first computed in the order $\mathbf{v}_1, \ldots, \mathbf{v}_s$ in line (6) and are then used in the same order in lines (10) and (11). Thus, the computation of $\mathbf{v}_{i+1}, \ldots, \mathbf{v}_s$ and the use of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}$ take place between the computation of vector $\mathbf{v}_i$ and the use of vector $\mathbf{v}_i$. A detailed analysis yields the access sequence $S_a(\mathbf{v}_i(j))$ and the access distance sequence $S_d(\mathbf{v}_i(j))$ for $j = 0, \ldots, n-1$ given in Table 1.

Table 1
Access sequence and access distance sequence for program version A

Line numbers	Access sequence $S_a(\mathbf{v}_i(j))$	Distance sequence $S_d(\mathbf{v}_i(j))$
(6)	$N_{i-1} + (4+3i)n + (j+1)(n+1)$	–
(4)	$N_i + (4+3(i-1))n + 3j + 2$	$N_i - N_{i-1} + d_1$
⋮	⋮	⋮
(4)	$N_{s-2} + (4+3(i-1))n + 3j + 2$	$N_{s-2} - N_{s-3}$
(10)	$N_{s-1} + 2n + 3j + 2$	$N_{s-1} - N_{s-2} + d_2$
(11)	$N_{s-1} + 5n + 3j + 2$	$3n$
Constants	$N_{-1} := 0$	$d_1 = 3j + 2 - 3n - (j+1)(n+1)$
used	$N_i = N_{i-1} + (4+3i)n + (n(n+1))$	$d_2 = 2n - (4+3(i-1))n$

Program A *(vector version):*

```
(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 = z1 + bbs[i] * v_i;
(11)     z2 = z2 + b[i] * v_i;
(12) }
(13) η_{κ+1} = η_κ + h* z2;
(14) err = h* z1;
```

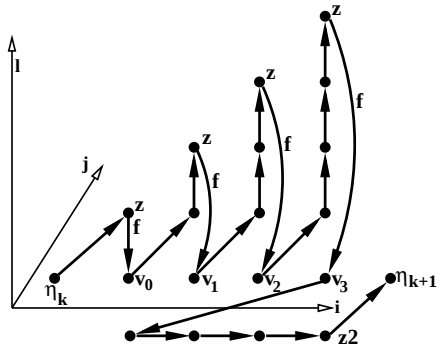

Fig. 2. Program version A (left) and corresponding computation order (right).

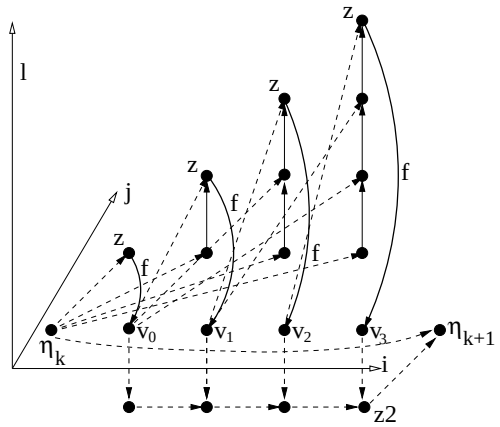

Fig. 3. Iteration space with dependence structure for an explicit RK method with $s = 4$. The filled circles depict entire vector components; the vector components in the j-direction are not shown explicitly. Broken lines represent data dependencies between corresponding vector components. Dependencies between entire vectors due to the evaluation of function **f** are shown by thick arrows annotated with **f**.

The numbers N_i are recursively defined and express the accumulated numbers of all accesses to all variables of the loop iterations i', $0 < i' < i$, before loop iteration i, $i < s$, starts. The access distances for $\mathbf{v}_i(j)$ grow according to the triangular iteration space, i.e. the distances increase proportional to $i \cdot n$ for $i = 0, \ldots s-1$. One time step contains $s - i + 2$ accesses to each component of stage vector $\mathbf{v}_i$ (consisting of $s - i$ accesses within the i-loop and two accesses for computing **z1** and **z2**). The sequences $S_a(\mathbf{v}_i(j))$ and $S_d(\mathbf{v}_i(j))$ are given for the case that **f** accesses all argument components (*dence* case); other access structures of **f** might result in different terms for the first components of $S_a(\mathbf{v}_i(j))$ and slightly different numbers N_i.

The sequence $S_d(\mathbf{v}_i(j))$ contains terms $N_{i+1} - N_i = (4+3i)n + n(n+1)$ which is a multiple of the dimension n (and is quadratic in n for the dense case) so that locality might be destroyed also for smaller system sizes. The large distances between accesses to the same component are caused by the interleaved access sequences $S_a(\mathbf{v}_i(j))$ to the stage vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{s-1}$, which is reflected by the occurences of $N_{i-1}, \ldots, N_{s+1}$ in all $S_a(\mathbf{v}_i(j))$.

Our final goal is to use a vector component as often as possible shortly after its computation. Due

to non-tightly nested loops and the dependencies described above, improving the temporal locality cannot be achieved by a simple restructuring of the computations but requires several intermediate transformation steps.

4.2. 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**: Separate argument vectors $\mathbf{z}[i]$ are introduced and replace $\mathbf{z}$ in the distinct stage vector computations in order to decouple the dependencies for later loop restructurings. 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 program fragment **B** in Fig. 4 where lines (1)–(6) replace the line (1)–(7) in program **A**.

The introduction of separate argument vectors **z**[i] increases the data set size but not the working set size since the new vectors are used in separate sections of the program execution:

$$dss(B) = (2s+5) \cdot n \quad wss(B) = (s+2) \cdot n.$$

The modified initialization leads to a slightly smaller working set size of $(i+1) \cdot n$ for loop trip i in program lines (1)–(6) and thus may reduce the number of conflict misses. On the other hand, the need to initialize more vectors may lead to more compulsory misses in line (2). It depends on the characteristics of the specific cache architecture which of these contrary modifications has a larger impact. The temporal locality behavior is similar to program **A**.

4.3. 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**. In principle, a loop interchange is now possible since the data dependencies between $\mathbf{z}$ and $\mathbf{v}_0, \ldots \mathbf{v}_{s-1}$ have been eliminated by the previous transformation. Starting with program version **B** several transformation steps have to be applied to lines (1)–(6) to get the equivalent program **C** with interchanged i-loop and l-loop.

First the independence of $\mathbf{z}[i]$, $i = 1, \ldots, s-1$ is exploited and the initialization in line (2) is separated from the computation in line (4):

```
for ( i=0; i<s; i++ )
   z[i]= η_κ;
for ( i=0; i<s; i++ ){
   for ( l=0; l<i; l++ )
      z[i] = z[i] + h* a[i][l] * v_l;
   v_i = f (x + c[i] * h , z[i]);
}
```

Next we concentrate on the second i-loop, which is a non-tightly nested loop. In this i-loop, the inner l-loop is empty for $i = 0$ and only the function evaluation for computing $\mathbf{v}_0$ is performed. The separation of this function evaluation from the nested loop is admissible since the argument vector $\mathbf{z}[0]$ is already available from the last time step. The second i-loop can therefore be replaced by:

```
v_0 = f(x + c[0] * h, z[0] );
for ( i=1; i<s; i++ ) {
   for ( l=0; l<i; l++ )
      z[i] = z[i] + h * a[i][l] * v_l;
   v_i = f(x + c[i] * h, z[i]);
}
```

This program fragment can be transformed into an equivalent loop nest in which a function evaluation for computing $\mathbf{v}_{i-1}$ is performed before the argument $\mathbf{z}[i]$ for the next function evaluation is computed. The last function evaluation for $\mathbf{v}_{s-1}$ appears after the nested loop:

```
for (i=1; i<s; i++)
   v_{i-1} = f(x + c[i-1] * h, z[i-1]);
   for (l=0; l<i; l++)
      z[i] = z[i] + h* a[i][l] * v_l;
}
v_{s-1} = f(x + c[s-1] * h, z[s-1]);
```

In the next step the function evaluation $\mathbf{v}_{i-1}$ is sunk into the inner l-loop by using a conditional so that this function evaluation is only computed once at the beginning of each l-loop:

```
for ( i=1; i<s; i++ )
   for ( l=0; l<i; l++ ) {
      if (l==0) v_{i-1} = f(x + c[i-1] * h , z[i-1]);
      z[i] = z[i] + h* a[i][l] * v_l;
   }
v_{s-1} = f(x + c[s-1] * h, z[s-1]);
```

The loop nest of the l-loop and the i-loop now forms a tightly nested loop. As loop-carried dependencies are preserved, the loops can be interchanged and the triangular iteration space is expressed by appropriate new loop bounds:

Program B *(separate argument vectors):*

```
(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 = z1 + bbs[i] * v_i;
(10)      z2 = z2 + b[i] * v_i;
(11)  }
(12)  η_{κ+1} = η_κ + h* z2;
(13)  err = h* z1;
```

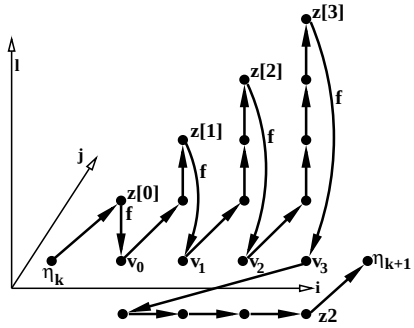

Fig. 4. Program version B with separate argument vectors (left) and corresponding computation order.

```
for ( l=0; l<s-2; l++ )
    for ( i=l+1; i<s; i++ ) {
        if (l==0) v_{i-1} = f(x + c[i-1] * h, z[i-1]);
        z[i] = z[i] + h* a[i][l] * v_l;
    }
v_{s-1} = f(x + c[s-1] * h, z[s-1]);
```

The inner function evaluation is only computed in the loop iteration $l = 0$ for $i = 1, \ldots, s-1$, i.e. the vectors $\mathbf{v}_0, \ldots, \mathbf{v}_{s-2}$ are computed in the first iteration of the l-loop. The loop body uses $\mathbf{v}_l$ so that $\mathbf{v}_l$ for $l = 0, \ldots, s-2$ has to be computed before, which can be performed in the same loop trip of l. Thus, it is possible to extract the function evaluation from the inner loop and to execute it as first statement of the l-loop. Including also the evaluation of $\mathbf{v}_{s-1}$ in the l-loop leads to the new loop bound $l < s$ (the inner i-loop is empty for $l = s-2$ and $l = s-1$):

```
for ( l=0; l<s; l++ ) {
    v_l = f(x + c[l] * h, z[l]);
    for ( i=l+1; i<s; i++)
        z[i] = z[i] + h* a[i][l] * v_l;
}
```

The resulting program fragment **C'** is shown in Fig. 5.

In program **C'** a stage vector $\mathbf{v}_l$ is first computed and then immediately used to partly build all argument vectors for subsequent function evaluations in the same time step. The data set and the working set remain unchanged. But the computations in lines (4)–(7) use only a working set of size of $s \cdot n$. The new loop structure avoids the interleaving of the accesses to different stage vectors in lines (3)–(7), so that the accesses to components of $\mathbf{v}_l$ are more concentrated. Also, the separation of the initialization of $\mathbf{z}[i]$ leads to an improved locality for accessing η_κ, since all accesses are now at the beginning of the time step which can be beneficial as η_κ has been computed at the end of the previous time step and might therefore still be in the cache. But the initialization and final computation of $\mathbf{z}[i]$ are now at more distant program points and the computation of $\mathbf{z}[i]$ is spread out over several iterations of the outer loop. Altogether, the positive and negative effect on the temporal locality nearly compensate each other and only slight effects on the execution time are expected, again depending on the details of a specific cache architecture. The analysis of the temporal locality results in the sequences of access times and access distances of $\mathbf{v}_i(j)$ for program version **C'** shown in Table 2.

The $s-l+2$ accesses to $\mathbf{v}_l(j)$ contain $s-l$ accesses not interleaved with accesses to other stage vectors, i.e., for all accesses w to $\mathbf{v}_l(j)$ and for all accesses w' to $\mathbf{v}_{l+1}(j)$ we have $w < w'$, since $w \leqslant N_l \leqslant w'$; the last two accesses to $\mathbf{v}_l(j)$ have larger distances.

Program version **C** results by merging the loop for computing $\mathbf{z1}$ and $\mathbf{z2}$ in line (9) with the l-loop in line (3) of program **C'**. All interleaved accesses to $\mathbf{v}_l$ are now removed. This is the condition required to use only one vector $\mathbf{v}$ to perform the computation of all stage vectors $\mathbf{v}_l$ for $l = 0, \ldots, s-1$ one after another:

Program **C** has the same data set size as the original program **A** but a larger working set size is needed in the first iteration of the l-loop:

$$dss(C) = (s+6) \cdot n \quad wss(C) = (s+3) \cdot n$$

The working set size decreases by n for each of the following iterations of the l-loop.

Program C corresponds to a delay of the function evaluations in computation scheme Eqs (2) and (3) so that a function evaluation is started not before its result is needed for another computation. The delay of

Table 2
Access sequence and distance sequence for program version C'

Line numbers	Access sequence $S_a(\mathbf{v}_i(j))$	Distance sequence $S_d(\mathbf{v}_i(j))$
(6)	$N_{l-1} + (j+1)(n+1)$	–
(4)	$N_{l-1} + n(n+1) + 0 \cdot 3n + 3j + 2$	d_1
(4)	$N_{l-1} + n(n+1) + 1 \cdot 3n + 3j + 2$	$3n$
⋮	⋮	⋮
(4)	$N_{l-1} + n(n+1) + (s-l-2)3n + 3j + 2$	$3n$
(10)	$N_{s-1} + 2n + 3j + 2$	$N_{s-1} - N_{l-1} + d_2$
(11)	$N_{s-1} + 5n + 3j + 2)$	$3n$
Constants used	$N_{-1} := 4n$ $N_l = N_{l-1} + (n(n+1)) + (s-l-1)3n$	$d_1 = (n-j-1)(n+1)$ $d_2 = (4-s+l)n - n(n+1)$

Program C' :

```
(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)  }
(8)  z1 = 0.0; z2 = 0.0;
(9)  for ( l=0; l<s; l++ ) {
(10)     z1 = z1 + bbs[l] * v_l;
(11)     z2 = z2 + b[l] * v_l;
(12) }
(13) η_{κ+1} = η_κ + h* z2;
(14) err = h* z1;
```

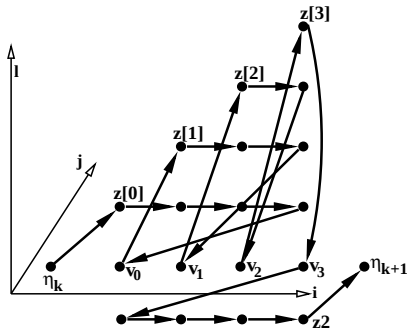

Fig. 5. Program version C' (left) and computation order (right).

Program C *(loop interchange):*

```
(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 = z1 + bbs[l] * v;
(7)      z2 = 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;
```

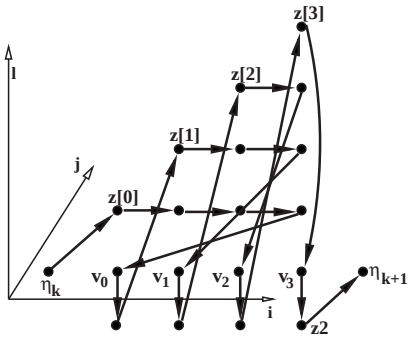

Fig. 6. Program version C (left) and computation order (right).

function evaluations can be achieved by applying the transformation $\mathbf{v}_i = \mathbf{f}(x_\kappa + c_i h_\kappa, \mathbf{w}_i)$ which results in the modified computation scheme:

$$\begin{aligned} \mathbf{w}_1 &= \eta_\kappa, \\ \mathbf{w}_2 &= \eta_\kappa + h_\kappa a_{21} \mathbf{f}(x_\kappa, \mathbf{w}_1), \\ &\vdots \end{aligned} \tag{4}$$

$$\mathbf{w}_s = \eta_\kappa + h_\kappa \sum_{i=1}^{s-1} a_{si} \mathbf{f}(x_\kappa + c_i h_\kappa, \mathbf{w}_i).$$

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

$$\eta_{\kappa+1} = \eta_\kappa + h_\kappa \cdot \sum_{l=1}^{s} b_l \mathbf{f}(x_\kappa + c_l h_\kappa, \mathbf{w}_l), \tag{5}$$

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

To avoid redundant computations, the results of the function evaluations are saved 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 Eq. (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.

4.4. Loop interchange with dimension loop

A further improvement of temporal locality is possible by a loop interchange with the innermost dimension loop. The innermost loop for the vector computation (now shown explicitly) 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 exhibits no interleaved use of different stage vector components so that the stage vector computations can be represented by a single scalar variable *fx* to represent all stage vector components of all stage vectors. This results in the following program fragment **D**:

Program D (*loop interchange with dimension loop*):

```
(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 = z1_j + bbs[l] * fx;
(8)           z2_j = 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;
(9)       }
(12)  }
(13)  η_{κ+1} = η_κ + h* z2;
(14)  err = h* z1;
```

The correctness of the replacement of $\mathbf{v}(j)$ by *fx* can also be seen from the access sequences. Before replacing all components $\mathbf{v}(j), j = 0, \ldots, n-1$, of $\mathbf{v}$ by the single variable *fx*, the sequence of access times for the j-th component $\mathbf{v}(j)$ is the concatenation

$$S_a(\mathbf{v}(j)) = S_a^0(\mathbf{v}(j)) || \ldots || S_a^{s-1}(\mathbf{v}(j))$$

of access sequences

$$\begin{aligned} S_a^l(\mathbf{v}(j)) = (&N_{l-1} + j(n + (s-l+1)3 + 1) \\ &+n+1, N_{l-1} + j(n + (s-l+1) \\ &3+1) + n + 3, \ldots, \\ &N_{l-1} + j(n + (s-l+1)3 + 1) \\ &+n + (s-l+1)3) \end{aligned}$$

where the numbers $N_i, i = -1, 0, \ldots, s-1$ are recursively defined as $N_{-1} := 6n; N_l = N_{l-1} + n(n+1) + (s-l+1)3n$ for $l = 0, \ldots, s-1$. For fixed $l \in \{0, \ldots, s-1\}$ all access times w to $\mathbf{v}(j)$ are smaller than access times $\bar{w}$ to $\mathbf{v}(j+1)$, i.e. $w < \bar{w}$, so that all components of $\mathbf{v}(j), j = 0, \ldots, n-1$ can be replaced by *fx*.

This kind of transformation cannot be applied to program **A** since a complete decoupling of vector computations is not possible. The reason is that the computation of the argument vector $\mathbf{z}$ could be changed in a similar style but that the entire vector $\mathbf{z}$ is still 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.

The transformation to program **D** reduces the data set size and the working set size to:

$$dss(D) = (s+5) \cdot n + 1 \quad wss(D) = (s+2)n.$$

Merging the loops over the vector elements to a single loop increases temporal locality. The access sequence of *fx* is

$$S_a(fx) \text{ with } S_a(fx) = S_a^0(fx) \parallel \ldots \parallel S_a^{s-1}(fx)$$

and a sequence $S_a^l(fx), l = 0, \ldots, s-1$ is the concatenation of the former sequences $S_a^l(\mathbf{v}(j))$, i.e.

$$S_a^l(fx) = S_a^l(\mathbf{v}(0)) \parallel \ldots \parallel S_a^l(\mathbf{v}(n-1))$$

The sequence of access distances consists of subsequences

$$S_d^{l,j}(fx) = (2, 3, \ldots, 3)$$

showing the good temporal locality. The distance between $S_d^{l,j}(fx)$ and $S_d^{l,j+1}(fx)$ is n and the distance between $S_d^{l,n-1}(fx)$ and $S_d^{l+1,0}(fx)$ is also n.

4.5. Constant coefficients for a specific RK method

Program versions **A**–**D** are formulated for general embedded RK methods, i.e., no specific RK coefficients are coded in the program. Instead the program versions contain references to arrays that are initialized with the coefficients of a specific RK method. This has the advantage of a large flexibility, but accessing the RK coefficients requires array accesses with the corresponding address calculations. For a fixed RK method these address calculations can be avoided by coding the RK coefficients as constants instead of array accesses. This requires an unrolling of all loops whose bounds depend on the number s of stages and whose body addresses the RK coefficients with the loop variable. For program version **D**, these are the outer l-loop and the inner i-loop. After the unrolling, each array access to the RK coefficients can be replaced by a constant with the corresponding value. We have performed this restructuring based on program version **D** for the popular DOPRIS method, leading to program version **E**.

5. Runtime experiments

We have implemented the RK program versions **A**–**E** and have investigated the resulting runtimes and the runtime improvements. As application problem we consider two example ODE systems.

5.1. Example applications

Brusselator equation The first example results from a spatial discretization of the Brusselator equation, a two-dimensional time-dependent partial differential equation describing a reaction-diffusion problem of two chemical substances [15]:

$$\frac{\partial u}{\partial t} = 1 + u^2 v - 4.4u + \alpha\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \quad (6)$$

$$\frac{\partial v}{\partial t} = 3.4u - u^2 v + \alpha\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)$$

for $0 \leqslant x \leqslant 1$, $0 \leqslant y \leqslant 1$, $t \geqslant 0$. The unknown functions u and v describe the concentrations of the two substances. A Neumann boundary condition $\frac{\partial u}{\partial n} = 0$ and $\frac{\partial v}{\partial n} = 0$ and the initial conditions $u(x, y, 0) = 0.5 + y, \quad v(x, y, 0) = 1 + 5x$ are used. A spatial discretization with N discretization points in each dimension based on the method of lines leads to an ODE system of size $n = 2N^2$:

$$\frac{dU_{ij}}{dt} = 1 + U_{ij}^2 V_{ij} - 4.4U_{ij} + \alpha(N-1)^2 (U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}, -4U_{i,j}) \quad (7)$$

$$\frac{dV_{ij}}{dt} = 3.4U_{ij} - U_{ij}^2 V_{ij} + \alpha(N-1)^2 (V_{i+1,j} + V_{i-1,j} + V_{i,j+1} + V_{i,j-1} - 4V_{i,j}) . \quad (8)$$

Each block of N^2 unknowns represents the concentration of one of the chemical substances. We use $\alpha = 2 * 10^{-3}$, which leads to a non-stiff ODE system for small and medium sizes of N. The ODE system becomes stiffer with increasing N, but an analysis of the stepsizes shows that an explicit RK method is still suited for the integration, see Fig. 7. A characteristics of the Brusselator system Eq. (8) is that each component of the right-hand side function $\mathbf{f}$ has a fixed evaluation time that is independent of the size n of the ODE system (sparse function), i.e., the evaluation time of the entire function $\mathbf{f}$ consisting of n components increases linearly with the size of the ODE system.

Schrödinger-Poisson equation The second example results from an application of a spectral method to a time-dependent 1D PDE describing the behavior of a collisionless electron plasma (Schrödinger–Poisson system) [20]:

$$i\hbar\frac{\partial \Psi}{\partial t} = -\frac{\hbar}{2m}\frac{\partial^2 \Psi}{\partial x^2} + e\Phi\Psi \quad (9)$$

$$\frac{\partial^2 \Phi}{\partial x^2} = \frac{e\, n_0}{\varepsilon_0}(1 - \Psi\Psi^*)$$

where L is the the length of the box in which N electrons are moving though a uniform positively charged background of density $n_0 = N/L$; e is the electron charge, m is the electron mass, and $i = \sqrt{-1}$. An L-periodic boundary condition $\Psi(0, t) = \Psi(L, t)$ in the space dimension is used. The normalization of the Schrödinger wave function Ψ is taken as $\int_0^L \Psi\Psi^* dx = L$ where Ψ^* is the conjugate complex of Ψ and the potential is $\Phi(x, t)$. The electron density is evaluated from the Schrödinger wave function as $n(x, t) = \Psi(x, t)\Psi^*(x, t)$. Using a Galerkin method with orthonormal functions $\{h_l\}_{l\in\mathbb{R}}$ and

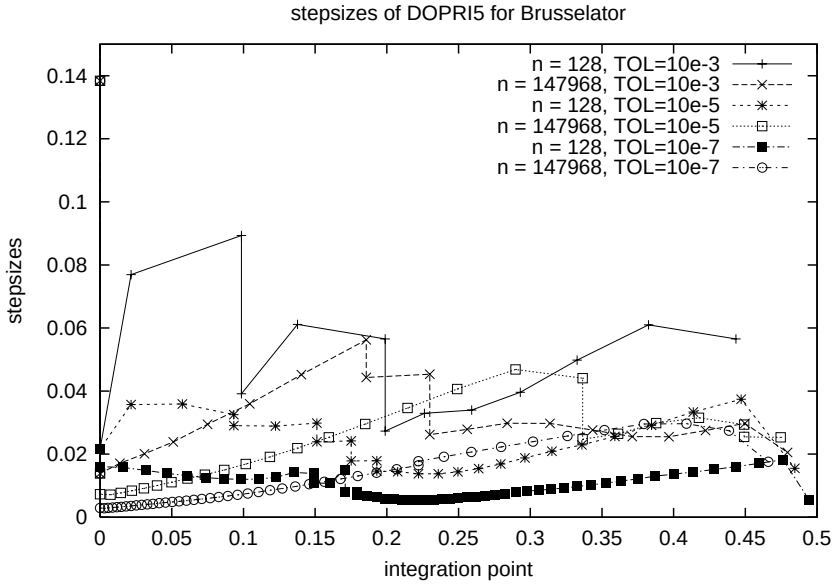

Fig. 7. Stepsizes used by DOPRI5 for different tolerance values for small and large Brusselator ODE systems.

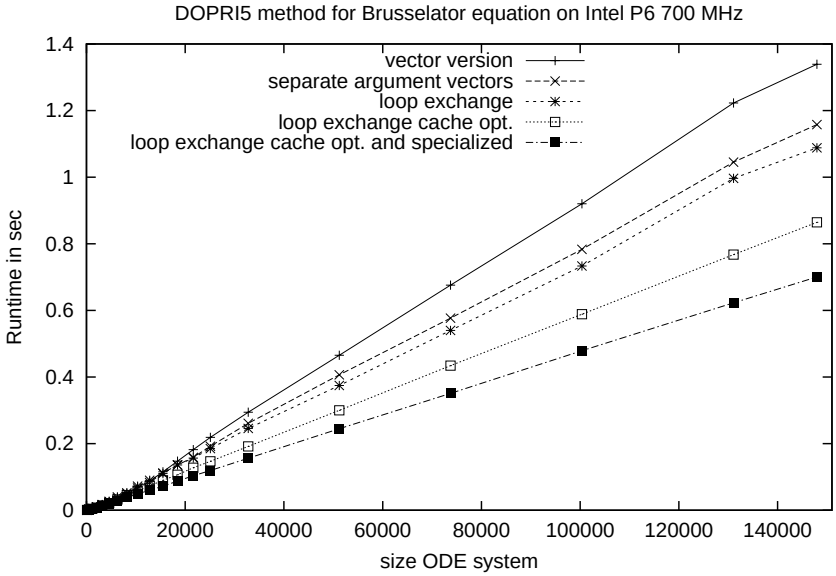

Fig. 8. Runtimes in seconds of the different RK versions on Pentium III, 700 MHz.

$$\psi_n = \sum_{|l|\leqslant n} \alpha_l(t) h_l,$$
$$\phi_n = \sum_{|l|\leqslant n} \beta_l(t) h_l, \qquad n \in N, \tag{10}$$

approximating the wave function ψ and the potential ϕ for the computation of the coefficients $\alpha_l = \alpha_l(t)$ results in the initial value problem:

$$i\frac{d\alpha_l}{dt} = -\lambda_l \alpha_l + \sum_{|j|\leqslant n} \alpha_l \cdot (\phi_n h_j, h_l),$$
$$l = 0, \pm 1, \ldots, \pm n \tag{11}$$

$$\beta_l = \frac{1}{\lambda_l}(|\psi_n|^2, h_l), \quad l = \pm 1, \ldots, \pm n \tag{12}$$

A characteristics of the Schrödinger system Eqs (11) and (12) is that the evaluation time of each component

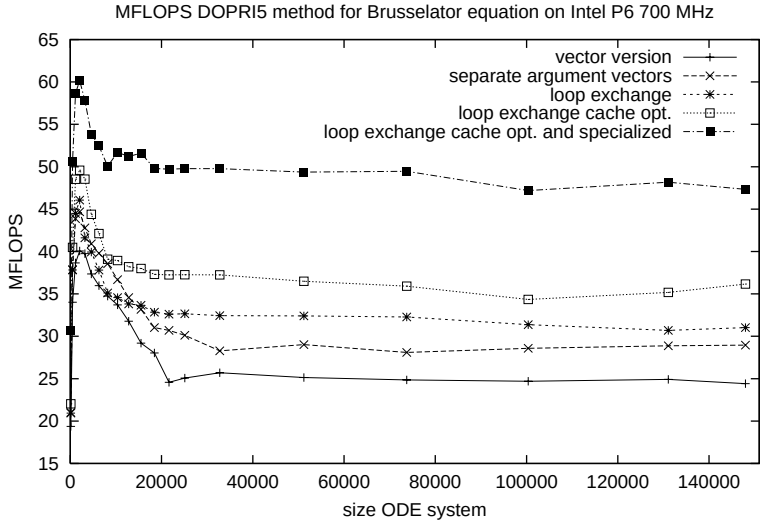

Fig. 9. MFLOPS of the different RK versions on Pentium III, 700 MHz.

of the right-hand side function **f** increases linearly with n, i.e., the evaluation time of the entire function **f** increases quadratically with the size of the ODE system (dense function).

5.2. Brusselator equation on Pentium III

As RK method we use the DOPRI5 method with $s = 7$ stages for both examples, as this is one of the most popular RK methods in practice. Similar results are obtained for the DOPRI8 method. The programs are written in C with double precision.

Implementation on Pentium III Fig. 8 shows the runtimes of all five program versions of the embedded RK method for different Brusselator ODE system sizes on a 700 MHz Pentium III processor (with 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; loop exchange cache opt. and specialized = Program E. The figure shows that each transformation leads to performance gains on the Pentium III. 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.

Figure 9 shows the corresponding MFLOPS rates which are obtained by the PCL library [3] 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 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 for the different versions can be analyzed by considering the cache miss rates.

The L1 cache miss rate in Fig. 10 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 since 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 11 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

Table 3
Overview of the cache architecture of the different processors used for the runtime experiments. The issue rate entry gives the number of instructions that can simultaneously be issued to different functional units of the CPU (maximum issue rate/number of MM instructions/number of integer instructions/number of floating point instructions/number of branch instructions)

Processor	L1 data cache	Latency L1	Cache line L1	L2 cache	Latency L2	Issue rate
Pentium 3	16 KBytes 2-way	3	32	16 KBytes 2-way	3	3/2/2/1/1
Power PC	64 KBytes 128-way	6–7	128	8 MBytes 4-way	36–52	4/2/3/2/1
Athlon	64 KBytes 2-way	3	64	256 KBytes 16-way	11	3/2/3/3/1
Alpha 21164	8 KBytes direct-mapped	2	32	96 KBytes 3-way	12	4/2/4/2/1
UltraSPARC II	16 KBytes direct-mapped	2	32	4 MBytes direct-mapped	4	4/1/2/2/1
MIPS RM 5200	16 KBytes 2-way	9	32	512 KBytes direct-mapped	15	2/1/1/1/1
MIPS R 10000	32 KBytes 2-way	10	32	4 MBytes 2-way	60	4/1/2/2/1
UltraSPARC III	64 KBytes 4-way	2	32	8 MBytes direct-mapped	15	4/1/4/3/1
Itanium 2	16 KBytes 4-way	1	64	256 KBytes 8-way	5–11	6/4/6/2/3
Xeon	8 KBytes 4-way	2	64	512 KBytes 8-way	14	3/2/3/2/1
Power4	32 KBytes 2-way	4	128 8-way	1440 KBytes	12	5/2/2/2/1

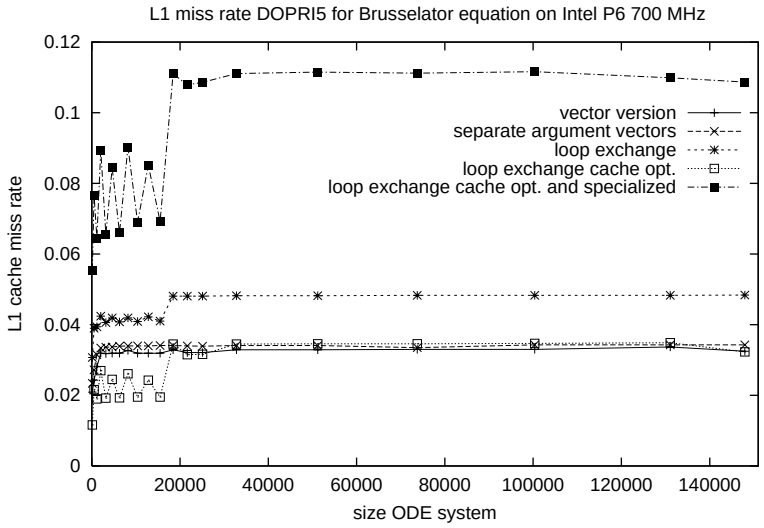

Fig. 10. L1 cache miss rate of the different RK versions on Pentium III, 700 MHz.

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 very 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.

In order to investigate the effect of the transformations on the runtime separately 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

Table 4
Runtime of the RK program versions in seconds for the Schrödinger function on a 2.0 GHz Intel Xeon

N	n	Program A	Program B	Program C	Program D	Program E
10	42	0.000204	0.000204	0.000204	0.000204	0.000250
20	82	0.000408	0.000306	0.000306	0.000306	0.000250
30	122	0.000918	0.000714	0.000612	0.000612	0.000500
40	162	0.001429	0.001020	0.001020	0.001020	0.001000
50	202	0.002041	0.001531	0.001429	0.001531	0.001500
60	242	0.002857	0.002143	0.002143	0.002041	0.002000
70	282	0.003776	0.002755	0.002653	0.002755	0.002750
80	322	0.004592	0.003571	0.003367	0.003469	0.003500
90	362	0.006020	0.004388	0.004286	0.004286	0.004250
100	402	0.007347	0.005408	0.005204	0.005306	0.005250

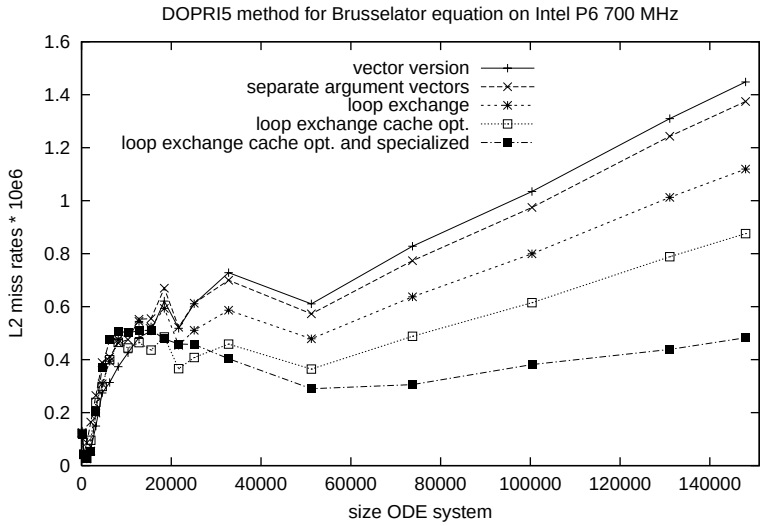

Fig. 11. L2 cache miss rate of the different RK versions on 700 MHz Pentium III. Actual values*10e6.

the specific application problem to be solved or the specific RK method to be used. So, the runtimes and MFLOPS measurements reflect the effect and improvement caused only by the transformations described in Section 4 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 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, since the implementation of **f** performs several index comparisons for each component of the argument vector of **f**. 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 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. The MFLOPS rate could be significantly increased by using a specialized version of the RK method that is adapted to the specific structure of the Brusselator equation, but this would not conform with the formulation of a black-box solver.

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 com-

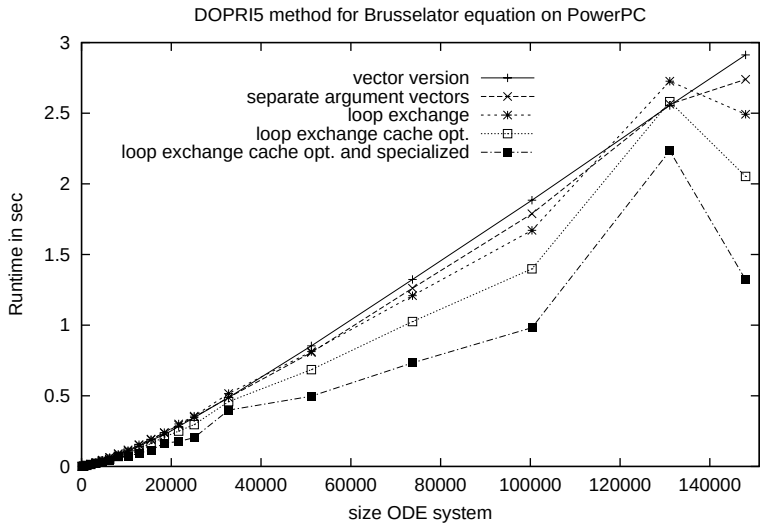

Fig. 12. Runtime of the RK program versions in seconds on a 300 MHz Power PC processor.

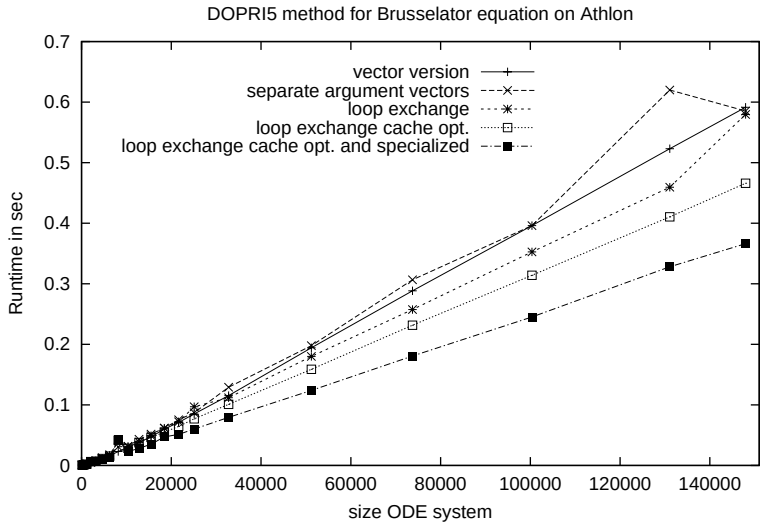

Fig. 13. Runtime of the RK program versions in seconds on a 700 MHz Athlon processor.

putation 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 program version **C** or **D**.

5.3. Brusselator equation on different processors

Figures 12–21 show the runtimes of one time step of the five RK implementations for the Brusselator ODE system 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 UltraSparcII processor, a 300 MHz MIPS/QED RM5200 processor, a 195 MHz MIPS R10000 processor, a 900 MHz Intel Itanium 2, a 750 MHz Sun UltraSparcIII, a 2.0 GHz Intel Xeon, and a 1.7 GHz IBM Power4. Table 3 gives an overview of the cache architecture of these processors.

The figures show that the proposed transformations can considerably reduce the runtime on recent processors like the AMD Athlon or the Intel Xeon processors with a similar cache structure as the Pentium III. On other processors the differences are sometimes smaller. Thus, the locality optimizing transformations are suit-

Table 5
Runtime of the RK program versions in seconds for the Schrödinger function on a 900 MHz Intel Itanium 2

N	n	Program A	Program B	Program C	Program D	Program E
10	42	0.001006	0.001036	0.000987	0.000957	0.000806
20	82	0.003019	0.003069	0.003029	0.002920	0.002734
30	122	0.006069	0.006138	0.006128	0.005929	0.005713
40	162	0.010164	0.010234	0.010334	0.009975	0.009790
50	202	0.015276	0.015346	0.015565	0.015047	0.014990
60	242	0.021445	0.021504	0.021903	0.021156	0.021240
70	282	0.028649	0.028679	0.029337	0.028300	0.028589
80	322	0.036850	0.036900	0.037767	0.036492	0.037036
90	362	0.046118	0.046138	0.047323	0.045689	0.046606
100	402	0.056451	0.056421	0.057936	0.055923	0.057227

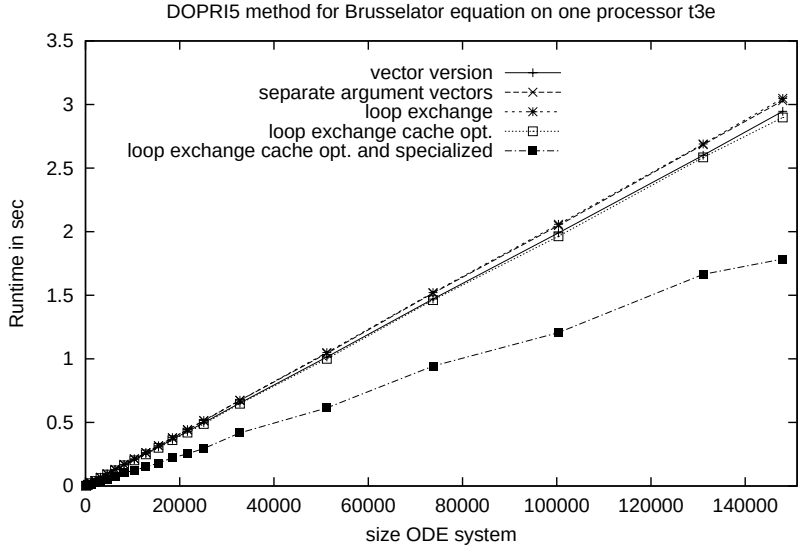

Fig. 14. Runtime of the RK program versions in seconds on DEC Alpha processor of the Cray T3E.

able 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 plays a less dominant role.

It can be observed that the improvements of program versions **D** and **E** are also obtained for small sizes of the ODE system. In fact, for most of the processors, the percentage improvement is larger for small systems than for large systems. For the Intel Xeon, for example, the execution time of program version **E** is 53% of the execution time of program version **A** for small ODE systems whereas for large ODE systems, program version **E** uses 68% of the execution time of **A**. A similar effect can be observed for the UltraSPARC III processor, where program version **E** reduces the percentage of the original execution time to 76% for small systems and to 82% for large systems. For the Intel Itanium on the other hand, the execution time of program version **E** is quite constant at 61% of the original execution time, independently from the size of the ODE system.

5.4. Tests for the Schrödinger example

Tables 4 and 5 show the execution times of one time step of the different RK versions for the Schrödinger equation. The results are shown as tables instead of figures because the quadratic dependence of the execution time on the system size leads to large differences in the execution time for different system sizes. The tables show that program versions D and E lead to the smallest execution times especially for small systems. For large systems, most of the execution time is needed for the evaluation of the function components which are not affected by the reorderings performed in the different program transformations, and thus the benefits of the evaluation reordering within the RK methods apply only to a small fraction of the execution time.

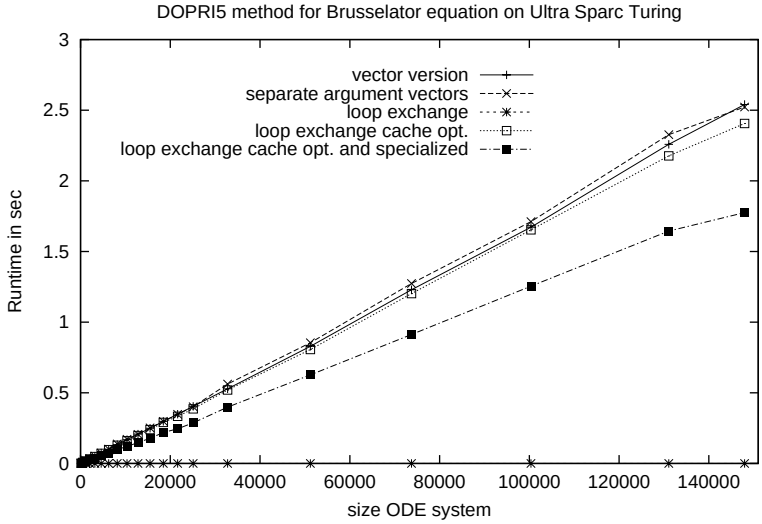

Fig. 15. Runtime of the RK program versions in seconds on a 300 MHz UltraSparc.

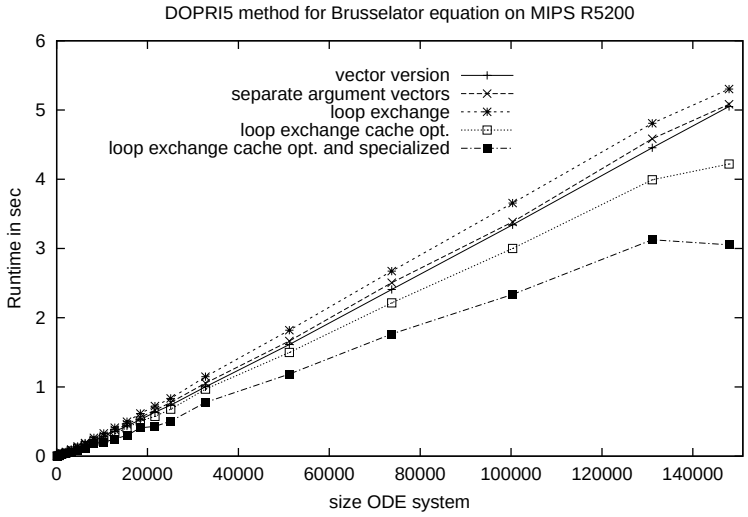

Fig. 16. Runtime of the RK program versions in seconds on a 300 MHz QED RM5200 processor.

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 [7] and iterative methods like 2D Jacobi [14] and multi-grid methods [24]. Many popular scientific libraries like LAPACK [2] 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) [4] and ATLAS (Automatically Tuned Linear Algebra Software) [26] 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

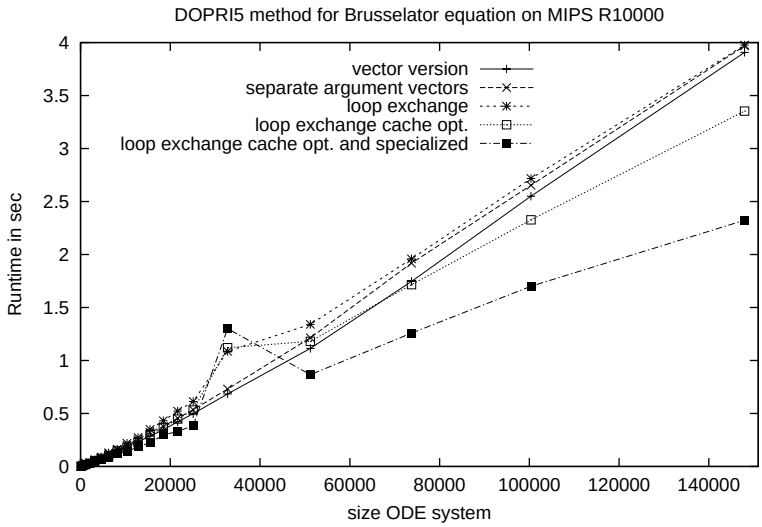

Fig. 17. Runtime of the RK program versions in seconds on a 195 MHz MIPS R10000.

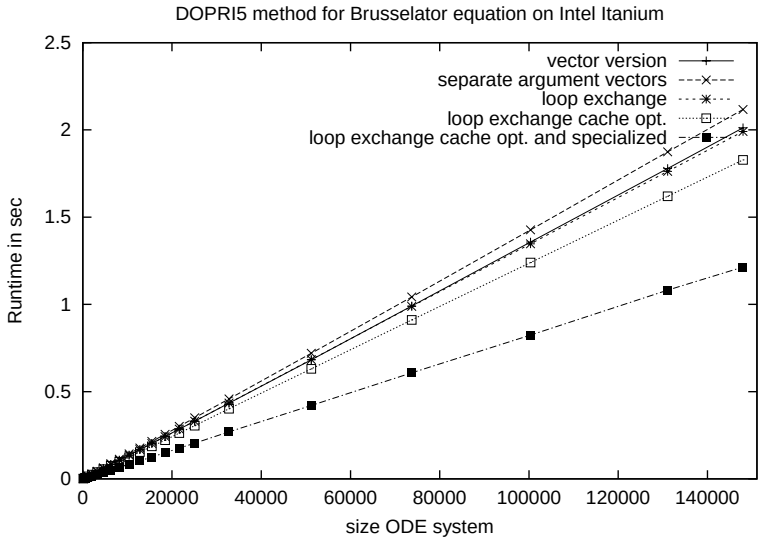

Fig. 18. Runtime of the RK program versions in seconds on a 900 MHz Intel Itanium 2.

loop unrolling factors by timings on the specific architecture. BLAS operations for larger arrays or matrices are build 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 [7,14] but not for ODE solvers. The cache performance of two- and three-dimensional multigrid algorithms is investigated in [25]. In particular, this paper considers a 2D red-black Gauss-Seidel relaxation method as the most time-consuming part of a multigrid method. The paper describes how the data accesses can be reordered to increase the temporal locality by using a blocking technique and array padding. The increased data locality leads to significantly larger MFLOPS rates especially for grids with a fine discretization and many grid points.

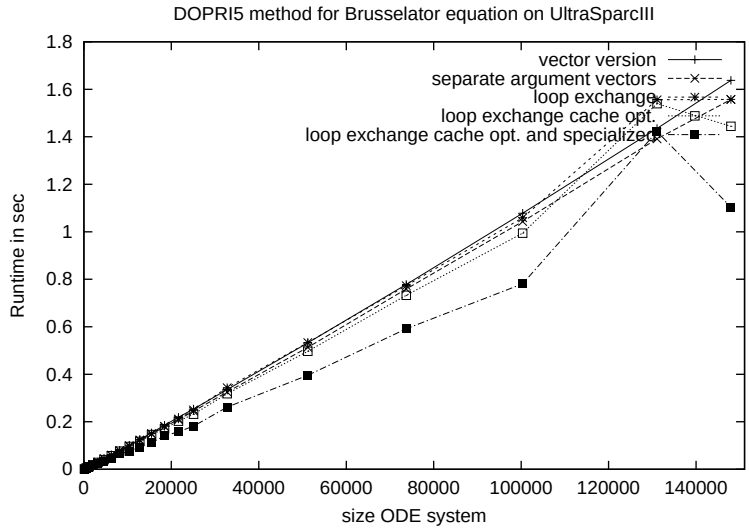

Fig. 19. Runtime of the RK program versions in seconds on a 750 MHz Sun UltraSPARCIII.

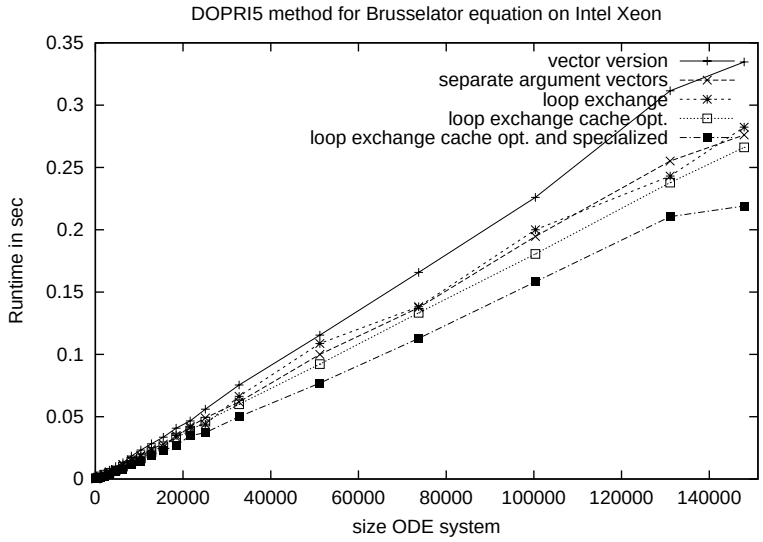

Fig. 20. Runtime of the RK program versions in seconds on a 2.0 GHz Intel Xeon.

7. Conclusions and future work

In this paper, we have investigated the locality behavior of embedded RK methods. We have presented a sequence of correctness preserving program transformations motivated by the goal to improve the temporal locality within each time step without affecting the good spatial locality of standard RK implementations. The transformation steps have been guided by the aim to improve the temporal locality of the stage vector computations which has required several intermediate transformations to make other transformations possible. A crucial assumption of the entire derivation has been to use only inherent properties of the embedded RK solvers for obtaining the intermediate transformations and the final code. This means in particular that the transformations have been developed without exploiting specific properties of the ODE problem to be solved or the specific cache architecture to be used. The experiments have shown that the resulting code leads to performance improvements for many recent processors and non-stiff ODE problems with different characteristics, especially for small and medium size ODE systems resulting from applying the method of

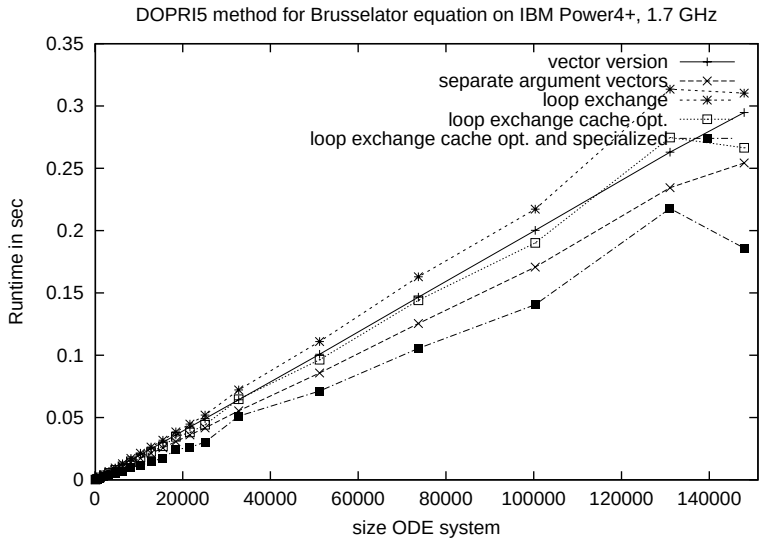

Fig. 21. Runtime of the RK program versions in seconds on a 1.7 GHz IBM Power4.

lines. Thus, the resulting modified RK code is ideally suited as library function for scientific libraries, which require a good overall performance. The performance improvements obtained depend on the characteristics of the ODE system to be solved. The experiments have shown that the performance improvements are much larger for sparse ODE systems than for dense ODE systems, since the execution time of dense ODE systems is dominated by the function evaluations which are not affected by the program transformations.

The locality optimizations presented in this paper lead to different lines of future work, including automatic support for the transformation process or investigations of other classes of ODE solvers. Other interesting classes of ODE solvers are implicit RK solvers for the solution of stiff ODE problems, but the computationally dominating part is usually the solution of the resulting (linear or non-linear) system of algebraic equations in each time step, leading to a completely different problem with different transformation requirements. Also, the locality optimizations of algebraic equation systems may require to give up the black-box assumption for the RK method.

Automatic or semi-automatic support is possible for the transformation process as described in this paper, since only standard transformations like loop fusion or loop interchange have been applied. Here, the strict separation of the optimization from the architectural details and to application problem is beneficial. The challenge lies in the selection of the order in which the transformations should be applied. A completely automated derivation of a cache-optimized version is therefore quite difficult. Appropriate measures should be invented and could be used to guide the program transformations. The program analysis based on access sequences is a first approach to quantify the effect of transformations which should be applied to a multitude of different numerical examples.

Acknowledgement

We thank the NIC Jülich for providing access to a Cray T3E and an IBM Regatta system and making the PCL library available. We also thank the anonymous referees for many helpful suggestions.

References

[1] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, 2002.

[2] 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.

[3] R. Berrendorf and B. Mohr, *PCL – The Performance Counter Library*, Version 2.0, Research Centre Jülich, September, 2000.

[4] 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.

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

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

[7] 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** (1996), 173–184.
[8] D. Culler, J.Singh and A. Gupta, *Parallel Computer Architecture*, Morgan Kaufmann, 1999.
[9] A. Darte, *On the Complexity of Loop Fusion*, in Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, IEEE, 1999, pp. 149–157.
[10] P.J. Denning, The Working Set Model for Program Behavior, *Communications of the ACM* **11**(5) (1968), 323–333.
[11] P. Deuflhard and F. Bornemann, *Scientific Computing with Ordinary Differential Equations*, Springer, 2002.
[12] W.H. Enright, D.J. Higham, B. Owren and P.W. Sharp, A Survey of the Explicit Runge-Kutta Method, Technical Report 94.291, University of Toronto, Department of Computer Science, 1995.
[13] E. Fehlberg, Classical fifth, sixth, seventh and eighth order Runge Kutta formulas with step size control, *Computing* **4** (1969), 93–106.
[14] K.S. Gatlin and L. Carter, *Architecture-cognizant divide and conquer algorithms*, in Proc. ACM/IEEE Supercomputing'99 Conference, 1999.
[15] E. Hairer, S.P. Nørsett and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, Springer Verlag, Berlin, 1993.
[16] F. Irigoin and R. Triolet, Supernode partitioning, in 15th Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif., January 1988, pp. 319–329.
[17] M. Kandemir, 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** (August 2000), 924–965.
[18] M. Kowarschik and C. Weiß, *An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms*, in Proc. of the GI-Dagstuhl Forschungseminar: Algorithms for Memory Hierarchies, Springer LNCS 2625, 2002.
[19] P.J. Prince and J.R. Dormand, High order embedded Runge-Kutta formulae, *J. Comp. Appl. Math.* **7**(1) (1981), 67–7.
[20] T. Rauber and G. Rünger, *Parallel Solution of a Schrödinger–Poisson system*, in International Conference on High-Performance Computing and Networking, Springer LNCS 919, 1995, pp. 697–702.
[21] T. Rauber and G. Rünger, Diagonal-Implicitly Iterated Runge Kutta Methods on Distributed Memory Machines, *Int. Journal of High Speed Computing* **10**(2) (1999), 185–207.
[22] T. Rauber and G. Rünger, Parallel Execution of Embedded and Iterated Runge Kutta Methods, *Concurrency: Practice and Experience* **11**(7) (1999), 367–385.
[23] G. Rivera and C.-W. Tseng, *Data Transformations for Eliminating Conflict Misses*, in Proc. of the ACM Conf. on Programming Language Design and Implementation, ACM, 1998.
[24] 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.
[25] C. Weiss, W. Karl, M. Kowarschik and U. Rüde, *Memory Characteristics of Iterative Methods*, in Proceedings of the ACM/IEEE Supercomputing'99 Conference, 1999.
[26] R.C. Whaley and J.J. Dongarra, *Automatically tuned linear algebra software*, Technical report, University of Tennessee, 1999.
[27] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison Wesley, 1996.