# Pipelining for Locality Improvement in RK Methods

Matthias Korch[1], Thomas Rauber[1], and Gudula Rünger[2]

[1] Universität Halle-Wittenberg, Institut für Informatik,
{korch,rauber}@informatik.uni-halle.de
[2] Technische Universität Chemnitz, Fakultät für Informatik,
ruenger@informatik.tu-chemnitz.de

**Abstract.** We consider embedded Runge-Kutta (RK) methods for the solution of ordinary differential equations (ODEs) arising from space discretizations of partial differential equations and study their efficient implementation on modern microprocessors with memory hierarchies. For those systems of ODEs, we present a block oriented pipelining approach with diagonal sweeps over the stage and approximation vector computations of RK methods. Comparisons with other efficient implementations show that this pipelining technique improves the locality behavior considerably. Runtime experiments are performed with the DOPRI5 method.

## 1 Introduction

Time-dependent partial differential equations (PDEs) with initial conditions can be solved by discretizing the spatial domain using the method of lines. This leads to an initial value problem (IVP) for a system of ODEs in the time domain of the form

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

where $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ is the unknown solution, $\mathbf{y}_0$ is the initial vector at start time $x_0$, $n \geq 1$ is the system size, and $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is the right hand side function describing the structure of the ODE system.

RK methods with embedded solutions are one of the most popular one-step methods for the numerical integration of non-stiff IVPs of the form (1). At each time step, these methods compute a discrete approximation vector $\eta_{\kappa+1} \in \mathbb{R}^n$ for the solution function $\mathbf{y}(x_{\kappa+1})$ at position $x_{\kappa+1}$ using the previous approximation vector $\eta_\kappa$. We consider an $s$-stage RK method that uses $s$ stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s \in \mathbb{R}^n$ with

$$\mathbf{v}_l = \mathbf{f}(x_\kappa + c_l h_\kappa, \eta_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li} \mathbf{v}_i) \ , \qquad l = 1, \ldots, s \ , \tag{2}$$

to compute two approximation vectors of different order according to:

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

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 724–733.

$\hat{\eta}_{\kappa+1}$ is an additional vector for error control. The $s$–dimensional vectors $b = (b_1, \ldots, b_s)$, $\hat{b} = (\hat{b}_1, \ldots, \hat{b}_s)$, $c = (c_1, \ldots, c_s)$ and the $s \times s$ matrix $A = (a_{li})$ specify the particular RK method. For non-stiff ODE systems of the form (1), explicit RK methods with an error control and stepsize selection mechanism are robust and efficient [6] and guarantee that the obtained discrete approximation of $\mathbf{y}$ is consistent with a predefined error tolerance [6,4].

In this article, we investigate the efficient implementation of RK methods on recent microprocessors. Modern microprocessors exhibit a complex architecture with multiple functional units and a storage hierarchy with registers, two or three caches of different size and associativity, and a main memory. Memory hierarchies provide improved average memory access times due to the locality of reference principle. As a consequence, spatial and temporal locality of a program have a large influence on the execution time.

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 [3] and iterative methods like 2D Jacobi [5] and multi-grid methods [8]. Based on BLAS, there are efforts like PHiPAC [2] and ATLAS [9] to provide efficient implementations of BLAS routines. Approaches for dense linear algebra algorithms or grid based methods are given in [3,5]. Locality optimizations for general purpose RK methods solving non-stiff ODEs with step-size control are investigated in [7]. In this paper, we consider ODE systems resulting from discretized PDEs which have a specific structure with a low coupling density induced by the coupling of the original PDE system and we investigate how this structure can be exploited to increase the locality of memory references. In particular, we use the specific access structure of $\mathbf{f}$ for a block-oriented pipelined computation of stage and approximation vectors. This is the basis for a reordering of the computation such that the working space of the algorithm is significantly decreased, which leads to a considerably better locality behavior. We show that these new computation schemes lead to significant reductions of the execution time on modern microarchitectures like the Pentium III or the UltraSPARC III processors.

## 2 Data Access Structure and Pipelining

For an embedded RK method applied to an ODE system of type (1), several equivalent program variants realizing different execution orders have been derived in [7] and the impact on the resulting execution time has been investigated for recent processors with memory hierarchies. The use of an arbitrary right hand side function $\mathbf{f}$ of (1) implies the conservative assumption that every component of $\mathbf{f}$ depends on all components of its argument vector and this limits the possibilities of code arrangements.

In this paper, we consider ODE systems (1) with a more specific right hand side function $\mathbf{f}$ depending on only a few components of the argument vector. Those ODE systems arise when applying the method of lines to a time-dependent PDE. As a typical example, we use a non-stiff ODE system resulting from a 2D

```
for (i = 0; i < s; i++) {
    for (j = 0; j < n; j++) w_i[j] = η[j];
    for (l = 0; l < i; l++)
        for (j = 0; j < n; j++) w_i[j] += h a_il v_l[j];
    for (j = 0; j < n; j++) v_i[j] = f_j(x + c_i h, w_i); }
for (j = 0; j < n; j++) { t[j] = 0.0; u[j] = 0.0; }
for (i = 0; i < s; i++)
    for (j = 0; j < n; j++) { t[j] += b̃_i v_i[j]; u[j] += b_i v_i[j]; }
for (j = 0; j < n; j++) { η[j] += h u[j]; ε[j] = h t[j]; }
```

**Fig. 1.** Basic program variant ($\tilde{\mathbf{b}} = \mathbf{b} - \hat{\mathbf{b}}$).

Brusselator equation, a reaction-diffusion equation for two chemical substances. A standard five-point-star discretization of the spatial derivatives on a uniform $N \times N$ grid with mesh size $1/(N-1)$ leads to an ODE system of dimension $n = 2N^2$ for the discretized solution $\{U_{ij}\}_{i,j=1,\ldots,N}$ and $\{V_{ij}\}_{i,j=1,\ldots,N}$, see [6]. We start our investigations with an implementation variant from [7] for which pipelining is possible if the access structure of $\mathbf{f}$ fulfills some requirement, see also Fig. 1.

*Basic Program Variant:* The RK method (2) and (3) is implemented in a straightforward way with a set of stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$ and a separate set of argument vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ for function $\mathbf{f}$ with $\mathbf{v}_i = \mathbf{f}(x_\kappa + c_i h_\kappa, \mathbf{w}_i)$, $i = 1, \ldots, s$, so that the loop structure has a minimal number of data dependencies which enables many loop restructurings. In the program code, the stages are computed successively. At each stage $i = 1, \ldots, s$ a nested loop is executed that computes the elements of the argument vectors $\mathbf{w}_i$ by calculating the weighted sum of the stage vectors $\mathbf{v}_j$, $j = 1, \ldots, i-1$. The working space of stage $i$ of this implementation consists of $i \cdot n$ stage vector elements, $n$ argument vector elements, and $n$ elements of the approximation vector.

For comparison, we include a specialized program variant from [7].

*Specialized Program Variant:* After applying the transformation $\mathbf{v}_i = \mathbf{f}(x_\kappa + c_i h_\kappa, \mathbf{w}_i)$, $i = 1, \ldots, s$, see [7], the loop structure is re-arranged such that only one scalar value is needed to store all stage vector components temporarily. Moreover, a specific RK method with fixed coefficients and fixed number of stages (DOPRI5) is coded with further optimizations like the unrolling of loops over stages.

**Storage Schemes.** We consider two different linearizations of the grid points $\{U_{ij}\}_{i,j=1,\ldots,N}$ and $\{V_{ij}\}_{i,j=1,\ldots,N}$. The **row-oriented** organization

$$U_{11}, U_{12}, \ldots U_{NN}, V_{11}, V_{12}, \ldots, V_{NN} \tag{4}$$

results in function components $f_l$ accessing argument components $l-N, l-1, l, l+1, l+N$, and $l+N^2$ ($l-N^2$), if available, for $l = 1, \ldots, N^2$ ($l = N^2+1, \ldots, 2N^2$). This is a typical access structure for grid-based computations. A specific disadvantage concerning the locality of memory references is that for

the computation of each component $f_l$ a component of the argument vector in distance $N^2$ is accessed. A **mixed row-oriented** organization

$$U_{11}, V_{11}, U_{12}, V_{12}, \ldots, U_{ij}, V_{ij}, \ldots U_{NN}, V_{NN} \tag{5}$$

stores corresponding components of $U$ and $V$ next to each other and results in function component $f_l$ accessing argument components $l-2N, l-2, l, l+1, l+2, l+2N$ (if available) for $l = 1, 3, \ldots, 2N^2 - 1$ and $l - 2N, l - 2, l - 1, l, l + 2, l + 2N$ (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 a distance equal to $2N$.

**Pipelining.** For the basic computation scheme with storage scheme (5), a pipelined computation based on a division of the stage, the argument and the approximation vectors into $N$ blocks of size $2N$ can be exploited. The computation of an arbitrary block $J \in \{1, \ldots, N\}$ of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ requires the corresponding block $J$ of $\mathbf{v}_s$, which itself depends on block $J$ of $\mathbf{w}_s$ and, if available, the neighboring blocks $J - 1$ and $J + 1$ of $\mathbf{w}_s$ because of the access pattern of $\mathbf{f}$. The computation of the blocks $J - 1$, $J$, $J + 1$ of $\mathbf{w}_s$ requires the corresponding blocks of $\mathbf{v}_{s-1}$. But these blocks cannot be computed before the computation of the blocks $J - 2$ to $J + 2$ of $\mathbf{w}_{s-1}$ is finished. Altogether, each block $J$ of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ depends on at most $\sum_{i=1}^{s}(2i + 1) = s(s + 1) + s = s(s + 2)$ blocks of $\mathbf{w}_1, \ldots, \mathbf{w}_s$ of size $2N$ and $\sum_{i=1}^{s}(2i - 1) = s(s+1) - s = s^2$ blocks of $\mathbf{v}_1, \ldots, \mathbf{v}_s$ of size $2N$, see Fig. 2 (left).

This dependence structure can be exploited in a pipelined computation order for the blocks of the stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$ and the argument vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ in the following way: the computation is started by computing the first $s + 1$ blocks of argument vector $\mathbf{w}_1$. Since the computation of component $(\mathbf{v}_1)_l$ requires the evaluation of $f_l(x_\kappa, \mathbf{w}_1)$ and since $\mathbf{f}$ has the specific access

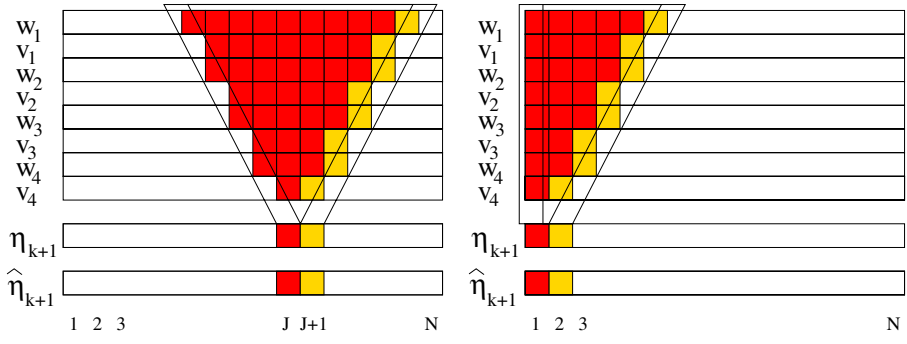



**Fig. 2.** *Left:* Dependence structure for storage scheme (5) in the case $s = 4$. If block $J$ of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ has been computed previously, the computation of block $J + 1$ requires accessing one additional block of each of the stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_4$ and the argument vectors $\mathbf{w}_1, \ldots, \mathbf{w}_4$ only. *Right:* Blocks accessed to compute the first and the second blocks of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$.

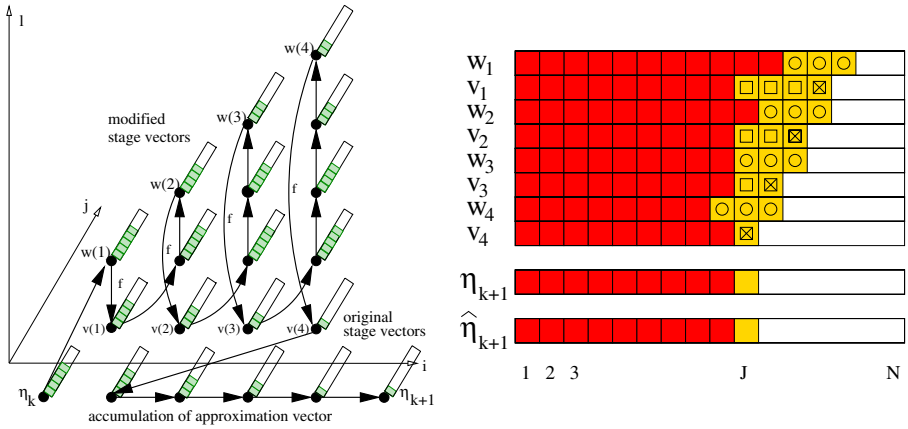



**Fig. 3.** *Left:* Illustration of pipelined computation for $s = 4$. The dimension of the vectors is shown to demonstrate the pipelined computation from Fig. 2. Filled boxes denote blocks of (intermediate) result vectors. The first block of $\eta_{\kappa+1}$ depends on all filled blocks shown in the figure. The filling structure of the vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ shows the triangular structure given in Fig. 2 (right). *Right:* Illustration of the working space of one pipelining step. Argument blocks marked by a circle are accessed during the function evaluation executed to compute the stage vector blocks tagged by a cross. Stage vector blocks used to compute blocks of argument and approximation vectors are marked by a square.

structure described above, the computation of $s$ blocks of $\mathbf{v}_1$ is enabled, which again enables the computation of $s$ blocks of $\mathbf{w}_2$ and so on. Finally one block of $\mathbf{v}_s$ is computed and used to compute the first block of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. The next block of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ can be determined by computing only one additional block of $\mathbf{w}_1$ which enables the computation of one additional block of $\mathbf{v}_1, \ldots, \mathbf{v}_s$ and $\mathbf{w}_2, \ldots, \mathbf{w}_s$, see Fig. 2 (right). This computation is repeated until the last blocks of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ are computed. Figure 3 (left) shows the iteration space of the pipelined computation scheme. The boxes attached to nodes illustrate the vector dimension of stage vectors and approximation vectors.

**Working Space.** The advantage of the pipelining approach is that only those blocks of the argument vectors are kept in the cache which are needed for further computations of the current step.

One step of the pipelining computation scheme computes $s$ stage vector blocks, $s$ argument blocks and one block of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. Since the computation of one block $J$ of one stage vector accesses the blocks $J-1$, $J$, and $J+1$ of the corresponding argument vector, altogether $3s$ argument blocks must be accessed to compute one block of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. Additionally, $\sum_{i=1}^{s} i = s(s+1)/2$ blocks of the stage vectors are accessed because the computation of one argument block $J$ requires the blocks $J$ of all previous stage vectors. Consequently, the working space of the pipelining computation scheme consists of $2+3s+s(s+1)/2$ blocks of size $2N$, see Fig. 3 (right). For the DOPRI5 method with $s = 7$ stages, at

most 51 blocks would have to be kept in cache to minimize the number of cache misses. This is usually a small part of 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

$$\frac{(2+3s+s(s+1)/2)}{(2s+2)N} = O\left(\frac{s}{4N}\right)$$

with usually $s \ll N$.

**Implementation.** The pipelining approach has been implemented in C in the following two program variants:

*Basic Pipelining:* The main body (Fig. 4 (a)) of the implementation consists of three phases: initialization of the pipeline, diagonal sweep over the argument vectors, and finalization of the pipeline. We introduce the following three macros (Fig. 4 (b)): STAGE0($A$) is used to compute one block of vector $\mathbf{w}_0$. Starting at offset $A$, STAGE($A$, $m$) computes one block of the stage vector $\mathbf{v}_{m-1}$ and one block of the argument vector $\mathbf{w}_m$. The macro FINAL($A$) evaluates the function values of one block of the last argument vector to obtain the corresponding stage vector block and finally computes one block of $\eta_{\kappa+1}$ and one block of the local error estimate $\epsilon_{\kappa+1} = \eta_{\kappa+1} - \hat{\eta}_{\kappa+1}$.

*Specialized Pipelining:* The second implementation is a pipelined version of the specialized implementation, which is optimized for a fixed number of $s = 7$ stages and exploits locality in the solution of Brusselator-like systems.

```
k = s · 2N;

for (j = 0; j < k; j += 2N)
    STAGE0(j);

for (i = 1, l = k − 2N; i < s; i++, l -= 2N)
    for (j = 0; j < l; j += 2N)
        STAGE(j, i);

for (j = k, l = k + 2N; j < n, j += l)
{
    STAGE0(j);
    for (i = 1, j -= 2N; i < s; i++, j -= 2N)
        STAGE(j, i);
    FINAL(j);
}

for (l = 1, k = n − 2N; l < s; l++)
{
    for (i = l, j = k; i < s; i++, j -= 2N)
        STAGE(j, i);
    FINAL(j);
}
FINAL(k);
```

(a) Body

```
STAGE0(A):
for (p = A; p < A + 2N; p++)
    w_0[p] = η[p];

STAGE(A, m):
for (p = A; p < A + 2N; p++)
    w_m[p] = η[p];
for (p = A; p < A + 2N; p++)
    v_{m−1}[p] = f_p(x + c_{m−1}h, w_{m−1});
for (r = 0; r < m; r++)
    for (p = A; p < A + 2N; p++)
        w_m[p] += h a_{mr} v_r[p];

FINAL(A):
for (p = A; p < A + 2N; p++) {
    v_{s−1}[p] = f_p(x + c_{s−1}h, w_{s−1});
    t[p] = 0.0; u[p] = 0.0; }
for (r = 0; r < s; r++)
    for (p = A; p < A + 2N; p++) {
        t[p] += b̃_r v_r[p];
        u[p] += b_r v_r[p]; }
for (p = A; p < A + 2N; p++) {
    η[p] += h u[p];
    ε[p] = h t[p]; }
```

(b) Macros

**Fig. 4.** Basic pipelining.

## 3 Runtime Experiments

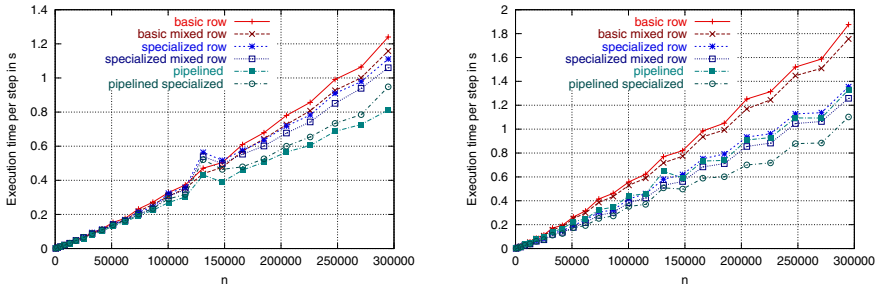
In this section, we investigate the performance enhancements achieved by the pipelining approach and compare the results with the general implementations for the two different storage schemes (4) and (5) of the Brusselator function. Different target platforms with varying memory hierarchies have been investigated:

1. UltraSPARC III at 750 MHz, 64 KB L1 data cache (4-way associative), 32 KB L1 instruction cache (4-way associative), 8 MB L2 cache (2-way associative),
2. UltraSPARC II at 450 Mhz, 16 KB L1 data cache (1-way associative), 16 KB L1 instruction cache (2-way associative), 4 MB L2 cache (1-way associative),
3. Pentium III at 600 MHz, 16 KB L1 data cache (4-way associative), 16 KB L1 instruction cache (4-way associative), 256 KB L2 cache (8-way associative),
4. MIPS R5000 at 300 MHz, 32 KB L1 data cache (2-way associative), 32 KB L1 instruction cache (2-way associative), 1 MB L2 cache.

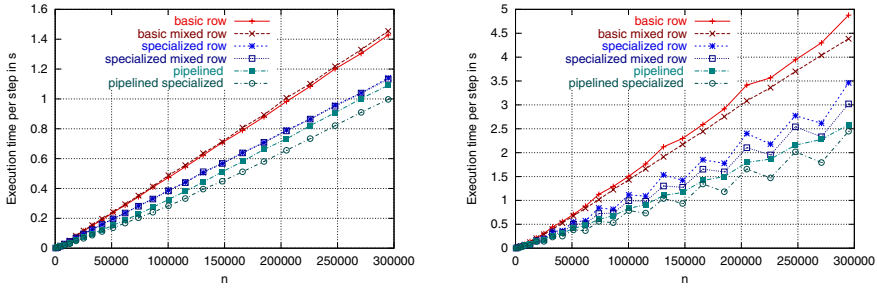
For the comparison, we present measurements for the basic version with storage scheme (4) (*basic row*) and storage scheme (5) (*basic mixed row*), for the specialized version with storage scheme (4) (*specialized row*), and with storage scheme (5) (*specialized mixed row*), for the pipelined version (*pipelined*), and for the pipelined specialized version (*pipelined specialized*). As RK method we use the DOPRI5 method with $s = 7$ stages. Figure 5 shows the execution times for one time step for the Brusselator equation on the target systems introduced above. Figure 6 shows the number of instructions executed and the L1 and L2 cache misses measured on the UltraSPARC III, and Fig. 7 shows those measurements for the Pentium III system. The data in Figs. 6 and 7 are obtained using the PCL library [1].

**Comparison of Storage Schemes.** Except for the Pentium III system, on all machines the mixed row-oriented storage scheme is significantly faster than the pure row-oriented scheme. The best results have been obtained on the MIPS processor. On this processor the use of the mixed row-oriented storage scheme leads to 10.13 % faster execution times for the basic implementation and 12.77 % for the specialized implementation when the size of the system is $n = 294\,912$. The execution times on the Pentium III system are very similar to each other for both storage schemes.

Figure 6 shows that the numbers of cache misses on the UltraSPARC III are very similar for both storage schemes. There are only slight improvements for the L2 and L1 data cache misses. The number of L1 instruction cache misses for the basic implementation with the mixed row-oriented storage scheme is even noticeably higher than the number measured with the original storage scheme. Thus, the improvements of the execution times achieved with the mixed row-oriented storage scheme on this machine seem to be caused by the lower number of instructions executed. The difference in the numbers of instructions executed for both storage schemes is caused by the different code of the function $\mathbf{f}$ and, as a consequence, the different number of machine instructions the two implementations of $\mathbf{f}$ are compiled to.




(a) UltraSPARC III (b) UltraSPARC II




(c) Pentium III (d) MIPS R5000

**Fig. 5.** Execution times of the RK implementations.

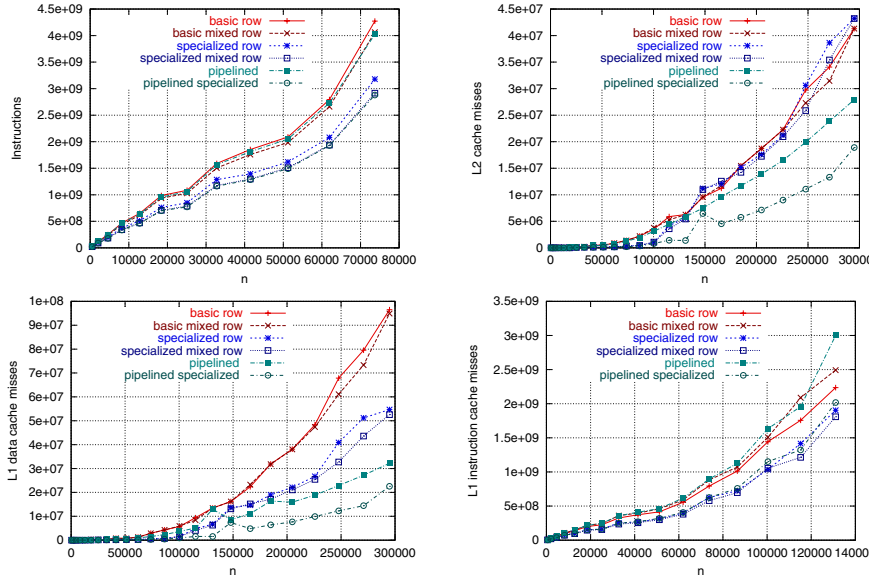



**Fig. 6.** Cache behavior and instructions executed on UltraSPARC III.

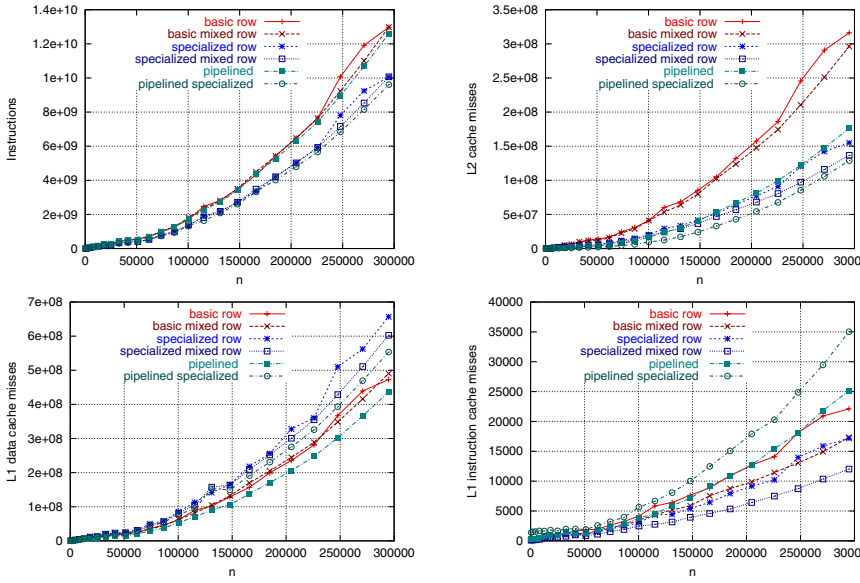



**Fig. 7.** Cache behavior and instructions executed on Pentium III.

On the Pentium III system the numbers of instructions executed for both storage schemes do not differ significantly. But the numbers of misses in the L2 cache and the L1 instruction cache are remarkably reduced for the mixed row-oriented storage scheme. The differences between the L1 data cache misses are smaller than those of the other caches. The basic implementation has even fewer L1 data cache misses with the pure row-oriented storage scheme for most system sizes.

**Pipelining.** The pipelining approach reduces the execution times on all machines we considered. Again the best results have been measured on the MIPS processor. For system size $n = 294\,912$, on this machine the basic pipelining implementation, which is specialized in the mixed row-oriented ordering, outperformed the basic general implementation by 41.00 %. The specialized pipelining implementation ran 18.94 % faster than the corresponding general implementation. On the other machines the basic pipelining implementation still was 23 % to 29 % faster than the basic general implementation, and the specialized pipelining implementation was 10 % to 12 % faster than the specialized general implementation. As expected, the enhanced locality of the pipelining approach leads to reduced L2 cache misses on the UltraSPARC III as well as the Pentium III processor. On the UltraSPARC III system the number of L1 data cache misses is also decreased. The number of L1 instruction cache misses does not change significantly on the UltraSPARC III but is increased on the Pentium III machine. Similarly, the number of instructions executed has hardly changed on the UltraSPARC III but is slightly smaller on the Pentium.

## 4 Conclusions

Runtime experiments have shown that for general RK implementations the mixed row-oriented storage scheme outperforms the row-oriented scheme on most of the processors considered. These results are due to higher locality caused by the smaller distance of the components accessed in one evaluation of the right hand side function **f**. Because of the increase in locality obtained by the pipelining computation scheme, we have measured reductions in execution time between 10 % and 41 %.

## References

1. R. Berrendorf and B. Mohr. *PCL - The Performance Counter Library, Version 2.0.* Research Centre Jülich, September 2000.
2. 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.
3. 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.
4. 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.
5. K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *Proc. of Supercomputing'99 Conference*, 1999.
6. E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems.* Springer–Verlag, Berlin, 1993.
7. Thomas Rauber and Gudula Rünger. Optimizing locality for ODE solvers. In *Proceedings of the 15th ACM International Conference on Supercomputing*, pages 123–132. ACM Press, 2001.
8. 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.
9. R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. Technical report, University of Tennessee, 1999.