Combining Measures for Temporal and Spatial Locality

Jörg Dümmler[1], Thomas Rauber[2], and Gudula Rünger[1]

[1] Chemnitz University of Technology, Department of Computer Science, 09107 Chemnitz, Germany {joerg.duemmler, ruenger}@cs.tu-chemnitz.de
[2] Bayreuth University, Angewandte Informatik II, 95440 Bayreuth, Germany rauber@uni-bayreuth.de

Abstract Numerical software for sequential or parallel machines with memory hierarchies can benefit from locality optimizations which are usually achieved by program restructuring or program transformations. The choice of the program version that achieves the best performance is usually complex as many dependencies have to be taken into account. Thus program-based locality measures have been introduced to give programmers a guideline if a performance gain can be expected from a program restructuring. The novel contribution of this paper is the extension of these locality measures to support spatial locality. These extended measures are applied to two applications from scientific computing and the obtained prediction is compared to benchmark results.

1 Introduction

Modern computer systems use a deep memory hierarchy including multiple levels of cache. Cache misses on these machines usually result in a waiting time of multiple clock cycles and can slow down applications considerably. Hence, the exploitation of the memory hierarchy provides the basis for an efficient execution of a given application. The number of cache misses is influenced by hardware specific parameters, e.g., the number of levels, the size and associativity of the cache, and software dependent parameters like the locality of the memory accesses. A high temporal locality is reached, if accesses to the same memory address lie closely together. An example is the repeated use of a scalar variable, e.g., to control the iterations of a loop. A high spatial locality is achieved, if accesses to neighboring memory locations lie closely together. An example is the consecutive use of the elements of an array variable.

Extensive research has been made to find program transformations which preserve the correctness of a program and increase the locality of the memory accesses. Applications from scientific computing usually spend a large proportion of the computing time in deeply nested loops. Therefore many transformations to increase memory locality for loop nests have been proposed. Examples include loop blocking or loop interchange, see [1] for a good overview. Deciding which program version achieves the best performance is a complex task and depends on the program code, the input data and on the target platform. In the general case not all this information is available, e.g. when parts of the considered program are not available in advance. Examples are solvers for ordinary differential equations (ODEs) which are usually written as black-box code that can operate on arbitrary ODE systems described by a function f.

Program-based locality measures, which allow the comparison of memory locality of different program versions, have been introduced in [2]. These measures only rely on the program source and do not take hardware dependent properties into account. Hence, an exact prediction of the cache misses is not possible as a memory access can result in a cache miss on one platform and a cache hit on another platform with a bigger cache. Nevertheless, the program-based locality measures can be used in an optimizing compiler tool or by a programmer to make a platform independent decision which program version to use or which program tranformations to apply. It has been shown that these measures can successfully capture the effects of temporal locality. The contribution of this paper is the extension of these measures with support for spatial locality. We study the extended cost measures for matrix multiplication and show the importance of spatial locality. As a more complex example we examine three different versions of an iterated Runge-Kutta ODE solver.

The rest of the paper is structured as follows. Section 2 introduces the program-based locality measures and suggests an extension for the support of spatial locality. Benchmark results for different program versions of a matrix-matrix-multiplication and a comparison with the predictions by the locality measures is discussed in section 3. Section 4 discusses related work and Section 5 concludes the paper.

2 Locality Measures Supporting Spatial Locality

The starting point of this work are the program-based locality measures presented in [2] which try to capture the memory access locality of a scientific application in a single value. The resulting value only depends on the program version and the input data size but is independent from platform specific characteristics. These measures observe each storage location in isolation and are therefore not able to uncover changes in spatial locality which plays an important role as we will show in section 3. This section introduces an extended definition of these cost measures which combines the effects of temporal and spatial locality.

To capture the effects of changes in spatial locality, accesses to physically neighboring memory locations have to be considered. Since we only take the program text as an input, it is not always possible to tell which of the variables are physically neighboring. The placement of the statically declared variables is subject to the compiler, which usually allocates neighboring storage locations to variables that are declared in adjacent positions in the program source. But most memory accesses are usually made to dynamically allocated memory, whose physical location is determined by the underlying operating system and cannot be predicted from the program source. As a consequence, we only consider the spatial locality of memory accesses which are made to the same data structure.

The locality measure is defined as a function $\mu : P \times \mathbb{N} \to \mathbb{R}^+$ where P is a set of equivalent program versions, $n \in \mathbb{N}$ is the input data size, and $\mu(A, n)$ is the locality value for $A \in P$ and input data size n. Lower values of this measure correspond to a better memory access locality, e.g. if $\mu(A, n) < \mu(B, n)$ for program version A and B then A is expected to have a better locality behavior than B for input data size n.

Let V_s be the set of variables of a program version A, where all variables, scalar or array, are represented by a single element $v_s \in V_s$. The total number of accesses to a variable $v_s \in V_s$ is denoted as $l_{v_s} + 1 \in \mathbb{N}$. The finite sequence $T = t_1, t_2, ...$ of consecutive natural numbers starting with $t_1 = 0$ represents the time indices of all memory accesses of a given program version. The *access sequence* of a given $v_s \in V_s$ is defined as the subsequence

$$n_0(v_s), n_1(v_s), ..., n_{l_{v_s}}(v_s) \subset T$$

of the sequence T. Furthermore, for array variables it is important to know, which element was referenced by a memory access. This information is stored in the *offset sequence* of a variable $v_s \in V_s$ that is defined as the sequence

$$o_0(v_s), o_1(v_s), ..., o_{l_{v_s}}(v_s) \quad \text{with } o_i(v_s) \in \mathbb{N}, \quad 0 \leq i \leq l_{v_s}.$$

For a scalar variable all elements of its offset sequence will be 0. We define the *spatial access distance* $d_{s_i}(v_s)$ of a variable $v_s \in V_s$ as

$$d_{s_i}(v_s) = \sqrt{(n_i(v_s) - n_{i-1}(v_s))^2 + (o_i(v_s) - o_{i-1}(v_s))^2}$$

with $1 \leq i \leq l_{v_s}$. In case of an array variable, the *spatial access distance* between two consecutive accesses to this variable decreases, if either the temporal distance is reduced or if the offsets of the elements involved lie closer together, i.e. the spatial locality is increased.

Figure 1 (left) shows an example for the computation of the *access distances* as defined for the temporal locality measures. Each element of the array has its own *access sequence* and therefore spatial locality between neighboring elements cannot be detected. In constrast the *spatial access distances* as defined in this paper combine temporal and spatial information as shown in Figure 1 (right). There is only one *access sequence* and an additional offset sequence for the array X in the example. The euclidian distance between two consecutive memory accesses to an array is used to computate the *spatial access distances.*

Based on the *spatial access distances*, the *average spatial access distance* of a variable $v_s \in V_s$ is defined as

$$M_s(v_s) := \left(\sum_{i=1}^{l_{v_s}} d_{s_i}(v_s) \right) / l_{v_s}$$

Following the definition of the temporal locality measures in [2] we define the following new cost measures which combine temporal and spatial locality:

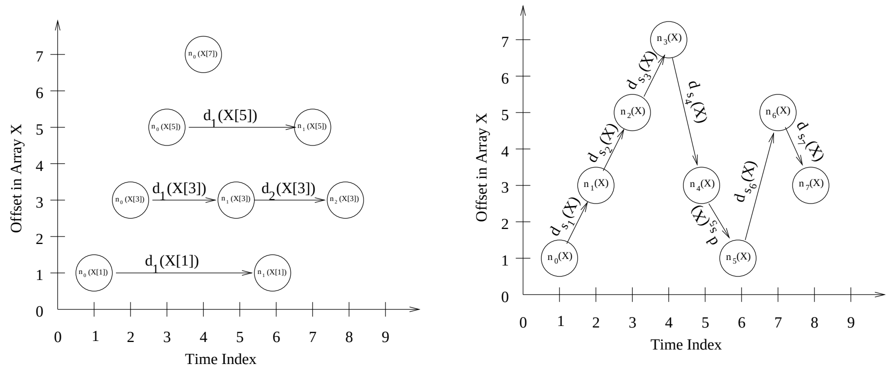


Figure 1. Example for the calculation of the *access distances* $d_i(v)$ as defined in [2] (left) and $d_{s_i}(v)$ as defined in section 2 (right).

Arithmetic mean of access distances:

$$\mu_{s_{AM}}(A,n) := \left(\sum_{v_s \in V_s} M_s(v_s) \cdot l_{v_s}\right) / \sum_{v_s \in V_s} l_{v_s},$$

Arithmetic mean of average access distances:

$$\mu_{s_{AA}}(A,n) := \left(\sum_{v_s \in V_s} M_s(v_s)\right) / \#V_s,$$

Sum of access distances:

$$\mu_{s_{SA}}(A,n) := \sum_{v_s \in V_s} \left(\sum_{i=1}^{l_{v_s}} d_{s_i}(v_s)\right),$$

Square of quadratic mean of access distances:

$$\mu_{s_{SQ}}(A,n) := \sum_{v_s \in V_s} \left(\sum_{i=1}^{l_{v_s}} d_{s_i}(v_s)^2\right) / \sum_{v_s \in V_s} l_{v_s},$$

Logarithmic geometric mean of access distances:

$$\mu_{s_{LG}}(A,n) := \sum_{v_s \in V_s} \left(\sum_{i=1}^{l_{v_s}} \log_2 d_{s_i}(v_s)\right).$$

Note that these definitions of locality measures do not cover spatial locality that may be exploited between different variables. This is only a disadvantage if the application uses many small arrays and scalar variables. Applications from scientific computing often operate on large data structures that are addressed using only a few pointer variables. In this case, spatial locality between different variables only plays a negligible role. Moreover, from the programmer's point of view, the placement of variables to memory locations cannot be directly influenced, i.e., the spatial locality within a single data structure is usually the target for program modifications.

3 Benchmark results

In this section, we present experimental results for different program versions and compare the resulting execution times to the predictions obtained by applying the locality measures. We use the multiplication of two square matrices as this is part of many applications from scientific computing and because the locality properties have been studied extensively. The program transformations used heavily rely on the exploitation of spatial locality. Therefore it is a good example to demonstrate the advantage of the extended measures. In the second example we study different program versions of an iterated RK solver for large ODE systems. These program versions use different computation schemes to calculate the argument vectors. The performance is influenced by temporal and spatial locality.

The program-based locality measures were calculated by utilizing a library in cooperation with a special simulation program. The simulation program mimics the memory access pattern of the target program version and calls special library function which accumulate all memory accesses and compute the cost measures. Memory accesses to index variables used for loop control were not considered, because, depending on the platform and the compiler, these variables are often stored in processor registers. In future versions a fully automated determination of the measures by a suitable compiler tool is planned.

The hardware characteristics of the platforms used for benchmark tests are summarized in Table 1.

Table 1. Platforms used for benchmark tests.

Processor	Intel Xeon	Intel Itanium 2	Intel Pentium 3	Sun UltraSparc III
Clock Rate	2.0 GHz	900 MHz	650 MHz	750 MHz
L1 Cache	8K data + 12K micro-ops	16K data + 16K instr., 4-way	16K data + 16K instr., 4-way	64K data + 32K instr., 4-way
L2 Cache	512K, 8-way	256 KB, 8-way	256 KB, 8-way	8 MB, 2-way
L3 Cache	n/a	1.5 MB, 12-way	n/a	n/a

3.1 Multiplication of two square matrices

As a first example we study different program versions for the multiplication of two matrices. Figure 2 (left) shows the pseudo code of a straight forward implementation. Through interchanging the loops in the second loop nest, six different program versions are derived, which are denoted as *mmm_xyz()*, where x is the index variable of the outermost loop, y the index variable of the middle loop and z the index variable of the innermost loop.

Assuming a row-wise data layout for all matrices, program version *mmm_ikj()* offers the best locality properties[1]. This is due to a stride 1 data access to the matrices B and C in the innermost loop, which results in an optimal exploitation of spatial locality. The accesses to matrix A can benefit from temporal locality in the innermost loop and from spatial locality in the middle loop.

Loop blocking is another popular program transformation, which can increase temporal locality. In each of the six program versions one, two or all three loops

function *mmm_ijk()*:

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    C[i][j] = 0;
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

function *mmm_ikj_3(bs)*:

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    C[i][j] = 0;
for (i = 0; i < N; i+=bs)
  for (k = 0; k < N; k+=bs)
    for (j = 0; j < N; j+=bs)
      for (ii = i; ii < min(i+bs, N); ii++)
        for (kk = k; kk < min(k+bs, N); kk++)
          for (jj = j; jj < min(j+bs, N); jj++)
            C[ii][jj] += A[ii][kk] * B[kk][jj];
```

Figure 2. Pseudo code of a matrix-matrix-multiplication with loop ordering (i, j, k) (left) and with loop ordering (i, k, j) and loop blocking applied to all three loops (right).

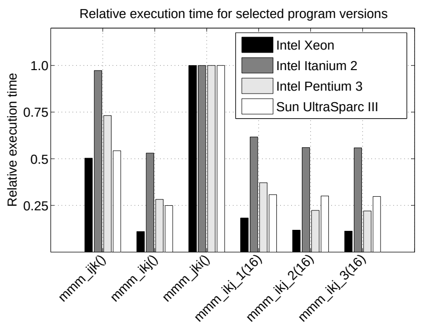


Figure 3. Relative execution time of program versions of a matrix-matrix multiplication using 1024x1024 matrices on different platforms.

can be blocked. This results in 18 additional program versions, which have the block size as a parameter. For simplicity we use the same block size for all loops. We denote these program versions as *mmm_xyz_n(bs)*, where x, y and z refer to the loop ordering, n gives the number of blocked loops and bs is the block size used. Figure 2 (right) gives the pseudo code of function *mmm_ikj_3(bs)*.

Figure 3 shows a selection of benchmark results scaled to a value of 1.0 for the slowest version, which was *mmm_jki()* in all cases. Considering the program versions without blocking, *mmm_ikj()* shows the best performance on all platforms. The speedup achieved depends on hardware characteristics, like the cache latency, and therefore differs from platform to platform. The block size leading to a minimum runtime is platform depend. We show a block size of 16, because it achieved competitive runtimes on all platforms.

Figure 4 shows on the left side the program-based locality measures for temporal locality as introduced in [2] and on the right side the extended cost measures introduced in this paper, both scaled to a maximum of 1. The temporal locality measures cannot predict the resulting runtime accurately, since temporal locality plays only a negligible role in the program tranformations. All spatial locality measures identify program version *mmm_jki()* as the one with the worst locality properties. The program versions with blocking and *mmm_ikj()* achieve the lowest spatial locality value and therefore are considered to have the best

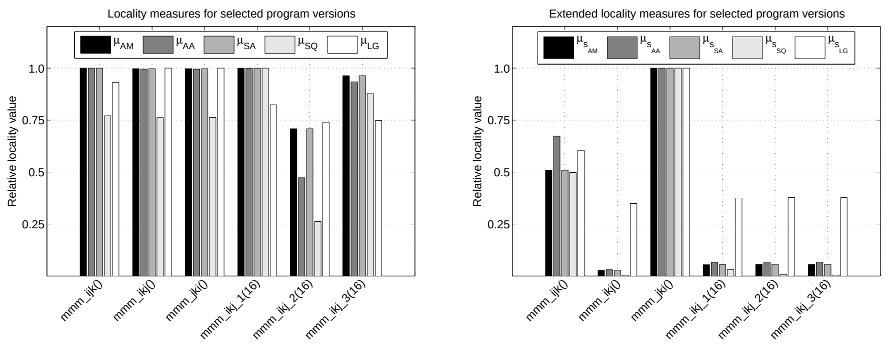


Figure 4. Relative values of the temporal locality measures (left) and the spatial locality measures (right) for different program versions of a matrix-matrix multiplication.

memory access locality. These results match with the runtime tests shown in Figure 3 and with the theoretical considerations in [1].

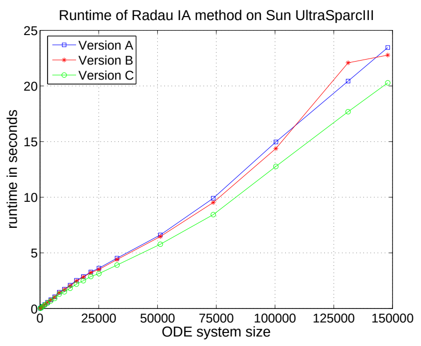
Altogether it can be stated, that the spatial locality measures are able to capture the locality properties of the different program versions of a matrix-matrix multiplication. These effects could not be uncovered using only the temporal locality measures.

3.2 Iterated Runge-Kutta methods

As a more complex example from scientific computing we study the spatial locality measures with different program versions of solvers for initial value problems (IVPs) of ODEs. Large systems of ODEs arise, e.g. when discretizing time dependent partial differential equations (PDEs) in the spatial domain using the method of lines[3]. Iterated RK solvers are explicit methods which were derived from classical implicit methods. The advantage of the iterated RK methods is the data independence of the computation of the stage vectors admitting a parallel execution [4]. From the implicit system of equations it is possible to construct an explicit system by computing approximations $\mu_l^{(i)}, i = 1, \dots, m$, for the stage vectors $v_l, l = 1, \dots, s$, using a fixed point iteration starting with η_k and using a fixed number of steps m, which depends on the RK method.

The following computation scheme shows the core of an iterated RK solver, where h is the step size and a_{ij}, b_i and c_i are parameters of the underlying implicit RK method:

```
for (l = 1; l <= s; l++)
  $\mu_l^{(0)}$ = $f(x_k,\ \eta_k)$;
for (i = 1; i <= m; i++)
  for (l = 1; l <= s; l++)
    $\mu_l^{(i)}$ = $f(x_k + c_l h,\ \eta_k + h_k \sum_{j=1}^{s} a_{lj} \mu_j^{(i-1)})$;
$\eta_{k+1}$ = $\eta_k + h_k \sum_{j=1}^{s} b_i \mu_j^{(m)}$;
```



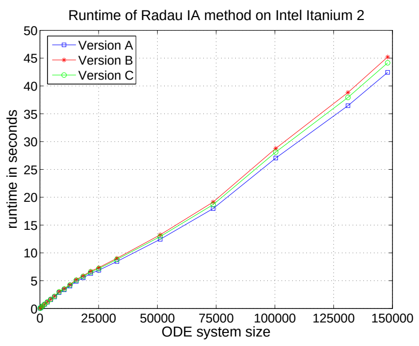
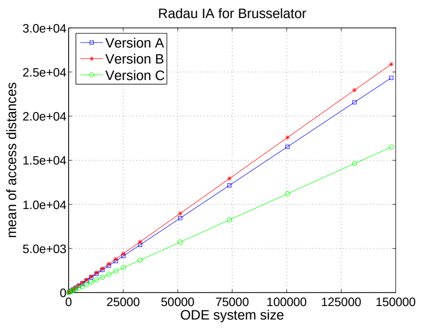


Figure 5. Runtime in seconds of a Radau IA method applied to the Brusselator ODE on a Sun UltraSparc III (left) and on an Intel Itanium 2 (right).



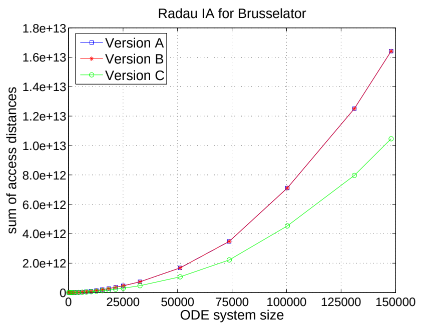


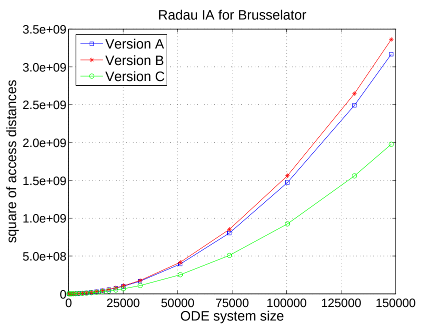
Figure 6. Locality measures $\mu_{s_{AM}}$ (left) and $\mu_{s_{SA}}$ (right) for Radau IA applied to the Brusselator ODE.

Version A: Program version A is a straightforward implementation of the computation scheme for iterated RK methods.

Version B: In this program version the computation of the argument vectors needed to calculate the approximation $\mu_l^{(j)}$ is modified. Separate argument vectors are introduced for each iteration, which results in a higher memory requirement and an additional multiplication per iteration. Some of the dependencies are resolved, so that further transformations are possible.

Version C: Through interchanging loops program version C is generated. Each computation is put in a separate loop nest, which results in an optimal exploitation of spatial locality. The computational and memory requirements are equal to those of version B.

To compare the performance of the three program versions we executed benchmark tests on different platforms. We use the Radau IA method [3] as basic RK method and solve the Brusselator equation [5] as example ODE system. The Brusselator equation is used to describe the reaction of two chemical substances with diffusion in the two dimensional space.



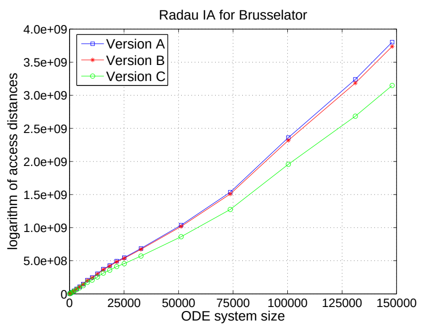


Figure 7. Locality measures $\mu_{s_{SQ}}$ (left) and $\mu_{s_{LG}}$ (right) for Radau IA applied to the Brusselator ODE.

Figure 5 shows the runtime for different system sizes on a Sun UltraSparc III processor and on an Intel Itanium 2 system. The performance of program versions **A** and **B** on the UltraSparc III platform are about equal, whereas program version **C** achieves an average speedup of 14%. On the Intel Itanium 2 the transformed program version **B** is 6% on the average slower compared to the original version **A**. The final program version **C** performs better than version **B** but cannot reach the performance of version **A**. This can be explained by the additional operations performed by versions **B** and **C** introduced by the transformation step.

Figure 6 (right) shows the spatial locality measure $\mu_{s_{SA}}$ for the program versions of an iterated RK solver. A similar result is obtained by applying measure $\mu_{s_{AA}}$. These two measures testify an about equal locality of memory accesses to program versions **A** and **B**. Program version **C** shows smaller locality values as expressed by the measures, i.e. is assumed to have a better memory access locality. The results obtained by cost measure $\mu_{s_{AM}}$ shown in Figure 6 (left) and by cost measure $\mu_{s_{SQ}}$ shown in Figure 7 (left) certify program version **B** a better memory access locality compared to version **A**. In contrast measure $\mu_{s_{LG}}$ presented in Figure 7 (right) yields a lower locality value and therefore a better memory access locality for program version **A**.

All spatial locality measures examined certify program version **C** the best memory access locality. The locality values of program versions **A** and **B** lie closely together for all measures. These result correspond with the measured runtimes on the Sun UltraSparc III processor very well. On the Intel Itanium 2 processor program version **A** achieves a smaller runtime. This program version requires fewer operations, which cannot be captured by the locality measures.

4 Related Work

An analytical examination of cache misses can be made by using cache miss equations, which can be used to calculate the position of cold misses and replacement misses in arrays. Direct mapped caches were analysed in [6] and the results generalized to associative caches in [7]. In [8] a worst case scenario is considered, which allows the prediction, whether a memory access is always a cache hit, always a cache miss or a prediction is not possible. The exact parameters of the memory hierarchy must be known to use this approach.

Cache misses for matrix-multiplication were analyzed in [9] for caches with different associativities and cache line sizes. In contrast, our approach tries to give an architecture independent measure of locality properties. An architecture-independent metric that represents the temporal behavior of data-movements of parallel programs in a distributed shared-memory environment has been presented in [10].

5 Conclusion and Future Research

In this paper we have introduced an extension to program-based locality measures which adds support for spatial locality. It has been shown that the spatial locality measures can be used to compare temporal and spatial locality properties of different program versions.

In future work we plan to add support for a fully automical determination of these measures. An extension of the cost measures to include the number of arithmetical operations is also possible. Another area of future research focuses on the extension of the cost measures for parallel programs.

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Academic Press, 522 B Street, Suite 1900, San Diego, CA 92101-4495, USA (2002)
2. Rauber, T., Rünger, G.: Program-based locality measures for scientific computing. International Journal of Foundations of Computer Science **15** (2004) 535–554
3. Hairer, E., Nørsett, S., Wanner, G.: Solving Ordinary Differential Equations. 2 edn. Volume 1. Springer (2002)
4. Rauber, T., Rünger, G.: Parallel Execution of Embedded and Iterated Runge-Kutta Methods. Concurrency - Practice and Experience **11** (1999) 367–385
5. Hairer, E., Wanner, G.: Solving Ordinary Differential Equations. 2 edn. Volume 2. Springer (2004)
6. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: An analytical representation of cache misses. In: Int. Conf. on Supercomputing. (1997) 317–324
7. Ghosh, S., Martonosi, M., Malik, S.: Cache miss equations: a compiler framework for analyzing and tuning memory behavior. ACM Transactions on Programming Languages and Systems **21** (1999) 703–746
8. Ferdinand, C., Wilhelm, R.: Efficient and precise cache behavior prediction for real-time systems. Real-Time Syst. **17** (1999) 131–181
9. Lam, M., Rothberg, E., Wolf, M.: The cache performance and optimizations of blocked algorithms. In: Proc. of the 4th Int. Conf. on Architectural support for programming languages and operating systems, ACM Press (1991) 63–74
10. Rodriguez, B., Jordan, H., Alaghband, G.: A Metric for the Temporal Characterization of Parallel Programs. Journal of Parallel and Distributed Computing **46** (1997) 113–124