Dynamic Selection of Implementation Variants of Sequential Iterated Runge–Kutta Methods with Tile Size Sampling

Natalia Kalinnik
University of Bayreuth
Department of Computer Science
95440 Bayreuth, Germany
natalia.kalinnik@uni-bayreuth.de

Matthias Korch
University of Bayreuth
Department of Computer Science
95440 Bayreuth, Germany
korch@uni-bayreuth.de

Thomas Rauber
University of Bayreuth
Department of Computer Science
95440 Bayreuth, Germany
rauber@uni-bayreuth.de

ABSTRACT

This paper describes an efficient self-adaptive procedure for iterated Runge–Kutta (IRK) methods, a class of solution methods for initial value problems (IVPs) of ordinary differential equations (ODEs). IRK methods execute a potentially large number of discrete time steps to compute the solution of the IVP. The performance of an IRK solver may strongly depend on the specific characteristics of the given IVP and the hardware architecture on which the solver is executed. To address this problem, this paper applies dynamic auto-tuning to the sequential execution of IRK methods.

Auto-tuning is a promising technique to avoid time consuming and extensive manual tuning. Our self-adaptive IRK solver utilizes the time-stepping nature of the IRK method. It selects the fastest implementation variant for the given IVP on the target architecture from a candidate pool during the first time steps. Then, the fastest implementation variant is used to compute all remaining time steps. The different implementation variants in the candidate pool have been developed by modifications of the loop structure of the basic algorithm. For those implementation variants that use loop tiling, we consider different tile sizes during the auto-tuning phase to further improve the performance of the self-adaptive IRK solver. Runtime experiments demonstrate the efficiency of the self-adaptive IRK solver for different IVPs on different hardware architectures.

Categories and Subject Descriptors

G.4 [**Mathematical software**]: Efficiency; G.1.7 [**Numerical Analysis**]: Ordinary Differential Equations—*Initial value problems, One-step (single step) methods*

ICPE'11, March 14–16, 2011, Karlsruhe, Germany.

General Terms

Algorithms, Experimentation, Measurement, Performance, Theory

Keywords

Auto-tuning, Tile Size Selection, Locality, Ordinary Differential Equations, Iterated Runge–Kutta methods

1. INTRODUCTION

The evolution of computer systems during the last decades has not only brought about systems with tremendous computational power, but also a great diversity of architectures with high complexity only understood by a small group of experts. The steadily increasing gap between the operation speed of processor chips and access times to off-chip memory has induced vendors to incorporate ever larger and deeper cache hierarchies. Therefore, on modern computer systems, the execution time of programs strongly depends on the specific order of the computations and memory accesses during the execution of the program. Program transformations like loop unrolling, loop tiling, and software pipelining can be used to modify the execution order of the program statements in order to better exploit the cache hierarchy and the processor architecture, thus leading to a significant reduction of program execution times [2, 1].

Due to the high complexity of modern computer architectures, applying these techniques manually to optimize for a specific architecture is challenging and time-consuming, because there are subtle interactions between the different transformations, the memory hierarchy of the target architecture and the dependencies in the source program. For programs to be run efficiently on a variety of architectures, this challenging and time-consuming optimization process has to be executed for every target architecture. In particular, once a manually tuned software has been shipped to the user, it cannot automatically adapt to new architectures on the market that the user might wish to buy.

What makes the situation even worse is that the memory access pattern of many commonly used programs depends on input data. Thus, to obtain maximum performance, such programs would have to be tuned for every possible set of input data separately. But this approach is not feasible, because usually the input data to be processed by the user is

not known in advance and the user cannot tune the software himself, because he is an expert in his application domain, but often does not have the expertise required to optimize software for modern complex hardware architectures.

Recently, these challenges have drawn the attention of many research groups. Several different approaches have been proposed to automatically tune software, but mainly in the field of linear algebra and signal processing (see Section 2). In this paper, we consider iterated Runge–Kutta (iterated RK or IRK) methods, a class of solution methods for initial value problems (IVPs) of ordinary differential equations (ODEs), which shares many characteristics with other time-step oriented algorithms.

Since the performance of the many different implementation variants of IRK methods possible strongly depends on the structure of the ODE system to be solved and on implementation parameters such as tile sizes for loop tiling, we propose a dynamically auto-tuned IRK solver based on the dynamic selection of implementation variants combined with the selection of suitable tile sizes. The resulting self-adaptive IRK solver exploits the time-stepping nature of IRK methods such that the evaluation of the implementation variants and tile sizes already contributes to the progress of the solution process. A detailed experimental evaluation shows that this approach is successful on different architectures and for ODE systems with different characteristics.

The remaining part of this paper is organized as follows: Section 2 presents an overview of existing approaches towards auto-tuning of computation-intensive applications. After this, Section 3 describes the time-stepping procedure and the computations performed by IRK methods. Section 4 reviews the implementation variants of IRK methods presented in [13], which constitute the candidate pool for the self-adaptive IRK solver. Then, Section 5 describes the realization of the self-adaptive IRK solver, which is evaluated by runtime experiments in Section 6. Section 7 concludes the paper.

2. RELATED WORK

Recently, several different approaches have been proposed to tune software automatically. Since loop tiling is considered as one of the most important program transformation techniques for locality improvement, many of these approaches target loop tiling. In addition, the automatic selection of implementation variants and algorithms also is one of the major goals.

In [10], the automatic generation of parametrically tiled parallel code from sequential C code is considered to enable an empirical search by auto-tuning software. The ATLAS library [22] optimizes basic linear algebra routines (BLAS) and some additional functions from LAPACK by generating a set of kernel routines with different tiling and unrolling parameters and selecting the best variant at installation time by feedback-driven empirical search. PHiPAC [3] takes a similar approach, but targets, in particular, matrix-matrix multiplication. [20] combines a compiler transformation framework with a parallel search algorithm. [24] presents an online tuning framework for runtime optimization and parallelization of Java programs for multi-cluster chip multiprocessors.

FFTW [7] is a software library for the computation of discrete fourier transforms (DFT) which uses a planner to adapt its algorithm to the hardware. The planner applies a set of rules to recursively decompose the problem into "sufficiently simple" equivalent subproblems. Before the computation begins, the planner selects a fragment of optimized code to be used for solving each of the resulting subproblems. SPIRAL [17] provides a program generation system for signal processing transforms such as DFT and discrete cosine transform that can generate implementations tuned for a specific target platform.

[23] shows that a model-driven approach can be as efficient as the empirical search used by ATLAS. [16] combines empirical feed-back directed exploration of loop structure choices with model-based tiling, parallelization, and vectorization. In [18], statistical machine learning is used to train an artificial neural network for building a model that can predict effective tile sizes. The SALSA project [6] uses machine learning to select an efficient solver, a suitable preconditioner, and further parameters for the solution of systems of linear equations based on a Bayesian classification of the input data.

3. ITERATED RK METHODS

IRK methods are a class of solution methods for ODE IVPs of the form

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad \mathbf{y}(t_0) = \mathbf{y}_0, \tag{1}$$

where $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is a given real-valued vector function (right-hand-side function), $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ is the unknown solution function, and $\mathbf{y}_0$ is the given initial value of the solution function at time t_0.

Numerical solution methods for ODE IVPs perform a time-stepping procedure consisting of a potentially large number of discrete time steps $\kappa = 0, 1, 2, \ldots$ corresponding to time t_κ. Starting at time t_0 with the initial approximation $\boldsymbol{\eta}_0 = \mathbf{y}(t_0) = \mathbf{y}_0$, at each time step κ a new approximation $\boldsymbol{\eta}_{\kappa+1}$ is computed using the approximation $\boldsymbol{\eta}_\kappa$ and maybe further previously computed approximations. The procedure repeats until the end of the integration interval $[t_0, t_e]$ is reached. Sophisticated methods estimate the local error committed at each time step and adapt the stepsize $h_\kappa = t_{\kappa+1} - t_\kappa$ according to the local error such that a specified accuracy is maintained and the number of time steps is reduced. Depending on the IVP to be solved, the total number of time steps required may lie between a few dozens and several millions or even more.

Many numerical methods for the solution of ODE IVPs exist. In addition to the classical explicit and implicit Runge–Kutta and multi-step methods [8], several methods with potential for parallelism such as IRK methods [14, 21], extrapolation methods [5, 8, 12], waveform relaxation methods [4], and peer two-step methods [19] have been proposed. IRK methods belong to the larger class of one-step predictor-corrector methods. Although they are derived from classical implicit RK methods, IRK methods are explicit methods: At each time step κ the new approximation $\boldsymbol{\eta}_{\kappa+1}$ is computed directly from the previous approximation $\boldsymbol{\eta}_\kappa$ by an iteration process; no (nonlinear) system of equations has to be solved.

The iteration process is started with an initial approximation for intermediate values of the solution function. We choose the "trivial" predictor:

$$\mathbf{Y}_l^{(0)} = \boldsymbol{\eta}_\kappa, \quad l = 1, \ldots, s. \tag{2}$$

Then, we continue the iteration process with a fixed number of $m = p - 1$ corrector steps

$$\begin{aligned} \mathbf{F}_i^{(k-1)} &= \mathbf{f}(t_\kappa + c_l h_\kappa, \mathbf{Y}_i^{(k-1)}), \\ \mathbf{Y}_l^{(k)} &= \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{s} a_{li} \mathbf{F}_i^{(k-1)}, \\ l &= 1, \ldots, s, \quad k = 1, \ldots, m, \end{aligned} \tag{3}$$

where $\mathbf{f}$ is the right-hand-side function of the ODE system to be solved. The coefficient matrix $A = (a_{li}) \in \mathbb{R}^{s,s}$, the weight vector $\mathbf{b} = (b_i) \in \mathbb{R}^s$, the node vector $\mathbf{c} = (c_l) \in \mathbb{R}^s$, and the order p are determined by the implicit RK method used as base method. The number s represents the number of stages.

After the iteration process, two approximations of different order, $\boldsymbol{\eta}_{\kappa+1}$ and $\hat{\boldsymbol{\eta}}_{\kappa+1}$, are computed as follows:

$$\begin{aligned} \boldsymbol{\eta}_{\kappa+1} &= \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{s} b_l \mathbf{F}_l^{(m)}, \\ \hat{\boldsymbol{\eta}}_{\kappa+1} &= \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{s} b_l \mathbf{F}_l^{(m-1)}. \end{aligned} \tag{4}$$

The norm of the difference of the two approximations

$$\epsilon = \| \boldsymbol{\eta}_{\kappa+1} - \hat{\boldsymbol{\eta}}_{\kappa+1} \|$$

yields an estimate of the local error committed in the current time step. Based on this error estimate, the stepsize can be adapted such that a larger step size can be chosen where a small stepsize is not needed to obtain the required accuracy, and, hence, the number of time steps and the computation time is reduced [8]. If ϵ is below a user-defined tolerance, the approximation $\boldsymbol{\eta}_{\kappa+1}$ is accepted and the algorithm proceeds with the next time step, generally using a larger stepsize. If, however, the new approximation does not satisfy the required accuracy, the stepsize control algorithm rejects the current time step and repeats it with a smaller stepsize.

4. CANDIDATE POOL

This section gives a short overview of several general and specialized implementation variants of IRK methods that constitute the candidate pool for the self-adaptive IRK solver to be presented in Section 5.

The implementation variants in the candidate pool differ in the loop structure of the corrector steps. The implementation of the corrector steps given by (3) leads to a nested loop structure with four dimensions iterating over:

1. the corrector steps ($k = 1, \ldots, m$),
2. the argument vectors $\mathbf{Y}_l^{(k)}$ ($l = 1, \ldots, s$),
3. the summands of $\sum_{i=1}^{s} a_{li} \mathbf{F}_i^{(k-1)}$ ($i = 1, \ldots, s$), and
4. the system dimension ($j = 1, \ldots, n$).

The candidate pool consists of general and specialized implementation variants. General implementation variants are suitable for arbitrary ODE problems, where the right-hand-side function $\mathbf{f}(t, \mathbf{y})$ may access all components of the argument vector $\mathbf{y}$. In this case, the evaluation of one component of the right-hand-side function $f_j\left(t_\kappa + c_l h_\kappa, \mathbf{Y}_i^{(k-1)}\right)$ may use all components of the argument vector $\mathbf{Y}_i^{(k-1)}$ and, thus, every argument vector component $Y_{l,j}^{(k)}$ to be computed in the current corrector step potentially depends on all argument vectors $\mathbf{Y}_1^{(k-1)}, \ldots, \mathbf{Y}_s^{(k-1)}$ computed in the previous corrector step. Therefore, the corrector steps have to be computed one after the other in the general case, and the k-loop has to be kept as outermost loop. All other loops with indices l, i, j are independent and fully permutable. Therefore, we can interchange, merge and split the l-, i- and j-loops, and loop tiling is also possible.

The specialized implementation variants in the candidate pool have been derived for special sparse ODE systems described by a right-hand-side function $\mathbf{f}$ that uses only a small number of components of the argument vector $\mathbf{y}$ to compute one component of the function result. For many sparse ODE systems, an ordering of the components can be chosen (e.g., by applying a bandwidth minimization scheme to the Jacobian of $\mathbf{f}$) such that the components of the argument vector accessed by the function evaluation are located within a limited index range near the component index evaluated. To measure this property of a function $\mathbf{f}$, we define the *access distance* $d(\mathbf{f})$ as the smallest value b, such that all component functions $f_i(t, \mathbf{y})$ access only the subset $\{y_{i-b}, \ldots, y_{i+b}\}$ of the components of the argument vector $\mathbf{y}$. We say the access distance of $\mathbf{f}$ is *limited* if $d(\mathbf{f}) \ll n$.

The specialized implementation variants exploit a limited access distance of the right-hand-side function $\mathbf{f}$ by overlapping of vectors and by a loop interchange of the j- and the k-loop leading to a pipeline-like computational structure of the corrector steps [13]. Thus, the locality behavior is improved by a significant reduction of the working space of the outermost loop, and storage space can be saved.

Table 1 summarizes all implementation variants in the candidate pool. A detailed description of the implementation variants is presented in [13].

5. SELF-ADAPTIVE IRK SOLVER

In this section, we describe an enhanced auto-tuning procedure for IRK methods. The procedure selects the fastest implementation variant for the current architecture and the IVP to be solved from the candidate pool dynamically. The general applicability of this auto-tuning approach to the time-stepping procedure of IRK methods was successfully demonstrated in [11]. This work further improves the previous auto-tuning approach by runtime selection of appropriate tile sizes.

5.1 Motivation

In the previous section, we have described a candidate pool consisting of several different general and specialized implementation variants of IRK methods. Generally, the execution time of these implementation variants varies depending on many factors such as the dimension of the ODE system considered and the access pattern of the right-hand-side function of the ODE system as well as the characteristics of the underlying hardware architecture. In particular, due to different cache parameters, such as size and associativity, it is not unusual that for a specific ODE system of given size an implementation variant performs best on one machine, but worst on another.

To substantiate our claims, we illustrate in Figure 1 the performance variation of the implementation variants in the candidate pool on two different computer systems. The first system is an AMD Opteron DP 246 machine running at

Implementation	Loop structure	Remarks
General implementation variants		
A	k–l–i–j	vector-oriented: inner loops iterate over system dimension; high spatial locality
E	k–l–j–i	exploits temporal locality of the i-loop, i.e., writes to argument vector components
D	k–i–j–l	exploits temporal locality of the l-loop, i.e., reads from results of function evaluations
Dblock	k–i–j–l–jj	similar to D, but loop tiling of the j-loop with the l-loop
PipeDe2m	k–j–i–l	based on D; j-loop surrounds l- and i-loop; exploits temporal locality of the i- and the l-loop
PipeDb2m	k–j–i–jj–l	similar to PipeDe2m, but loop tiling of the j-loop with the i-loop
PipeDb2mt	k–j–i–(jj)–l–jj	similar to PipeDb2m, but loop tiling expanded to the l-loop
Specialized implementation variants		
PipeDb1m	k–j–i–jj–l	similar to PipeDb2m, but the vectors $\mathbf{Y}_l^{(k)}$ are overlapped to reduce space requirements
PipeDb1mt	k–j–i–(jj)–l–jj	similar to PipeDb1m, but loop tiling expanded to the l-loop
ppDb1m	j–k–i–jj–l	based on PipeDb1m; j- and k-loop are interchanged using a pipelining approach
ppDb1mt	j–k–i–(jj)–l–jj	similar to ppDb1m, but loop tiling expanded to the l-loop

Table 1: Candidate pool of implementation variants.

2.0 GHz with 64 kB L1 cache and 1 MB L2 cache. The second system is an AMD Opteron 8350 with four quad-core processors running at 2.0 GHz and equipped with 64 kB L1 cache/core, 512 kB L2 cache/core and 2 MB shared L3 cache. On both systems, GCC 4.4.3 with optimization level 2 was used to compile the implementations. As example problem we choose the 2D Brusselator equation (BRUSS2D) [8] (see also Section 6.1). Since BRUSS2D can be implemented such that the right-hand-side function $\mathbf{f}$ has a limited access distance, all general and specialized implementation variants in the candidate pool are applicable. As corrector method we use the 3-stage method Radau IA (5) [8].

The figure shows execution time per time step and component as a function of the size of the ODE system, n. Since, for the test problems considered in this article, the computations performed for each component are independent of n, each implementation variant would produce a nearly flat line on computers with constant memory access time. On modern computers, caches lead to varying memory access times, and, since cache utilization changes as the system size grows, we observe variations of the normalized runtimes when we change the system size.

Considering the runtimes shown in Figure 1, we can make the following observations:

- The runtimes of the fastest and the slowest implementation variants differ by up to 20 %.
- On the same machine, but for different sizes n of the ODE system, the order of the implementation variants varies. In Figure 1, we can identify three sub-ranges, where different implementation variants offer the best performance.
- On different machines, for the same size n of the ODE system, different implementation variants obtain the best performance.

Most of the implementation variants in the candidate pool use tiling as an optimization technique to improve spatial and temporal locality. Loop tiling partitions the iteration space of a loop into smaller blocks (tiles) with the objective to keep the data accessed by successive loop partitions in cache. This leads to a better cache utilization and reduces the number of cache misses. Thereby, choosing suitable values for the tile sizes is an essential step to achieve good performance.

Tile size selection is a challenging task. If the tile size is chosen too small, the loop control overhead may outweigh the benefit of tiling. On the other hand, the tile size should not be chosen too large to guarantee that the resulting working space, i.e., the set of data elements referenced by processing the tile, still fits in the cache. The problem of selecting appropriate tile sizes is getting even more challenging by the fact that, depending on the dimension of the ODE system and the characteristics of the hardware architecture the application is executed on, the optimal tile size may vary significantly.

The contour plots in Figure 2 highlight the variation of the ranges of tile sizes with good performance on the AMD Opteron DP 246 system and on the AMD Opteron 8350 system. As example we choose again BRUSS2D. In the contour plots, the relative runtimes for implementation PipeDb2mt are plotted w.r.t. the best runtime over the range of tile sizes for different system sizes n, so that the color scale indicates the quality of the tile sizes. Blue regions in the contour plots are associated with good choices for the tile size, whereas red regions correspond to ranges with less appropriate tile sizes. The deviation in the performance for the range of tile sizes of $[1, 5000]$ considered is $\lesssim 10\,\%$ on both computer systems. A similar performance deviation has been observed for this range of tile sizes on other hardware architectures. For very large tile sizes, i.e., tile sizes close to n, the relative performance loss can be significantly higher. For the AMD Opteron DP 246, we observe that the selection of small tile sizes $\lesssim 700$ turns out to be the optimal strategy. In contrast, for the AMD Opteron 8350 the choice of small tile sizes may negatively affect the performance. For the range of tile sizes considered in Figure 2, the best performance for implementation PipeDb2mt on the AMD Opteron 8350 is observed for tile sizes close to 500 and in the range of $[3000, 5000]$.

We conclude that the performance of IRK solvers and similar time-stepping algorithms is quite sensitive to the choice of implementation variant and tile size as well as the characteristics of the hardware architecture. In order to address these issues, it is important to develop a self-adaptive procedure for IRK methods which can dynamically determine the fastest implementation variant and select an appropriate tile size for the given ODE system on the target architecture automatically.

Since the differences in the performance observed for different implementation variants are even higher than the differences in the performance observed for different tile sizes, the priority of an adaptive solver should be the selection of an efficient implementation variant, but a suitable tile size

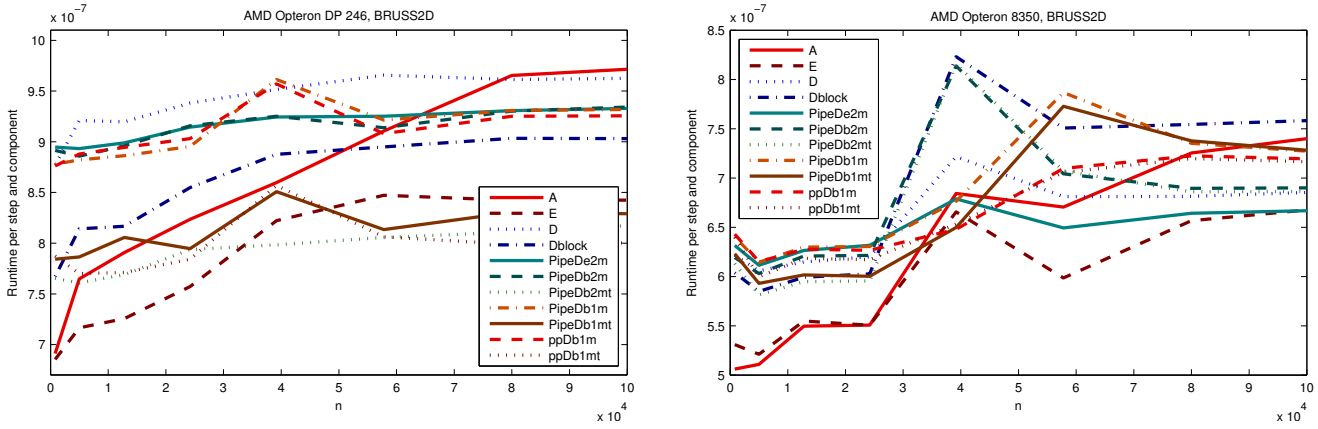

Figure 1: Normalized execution times for varying problem sizes of BRUSS2D for Radau IA (5).

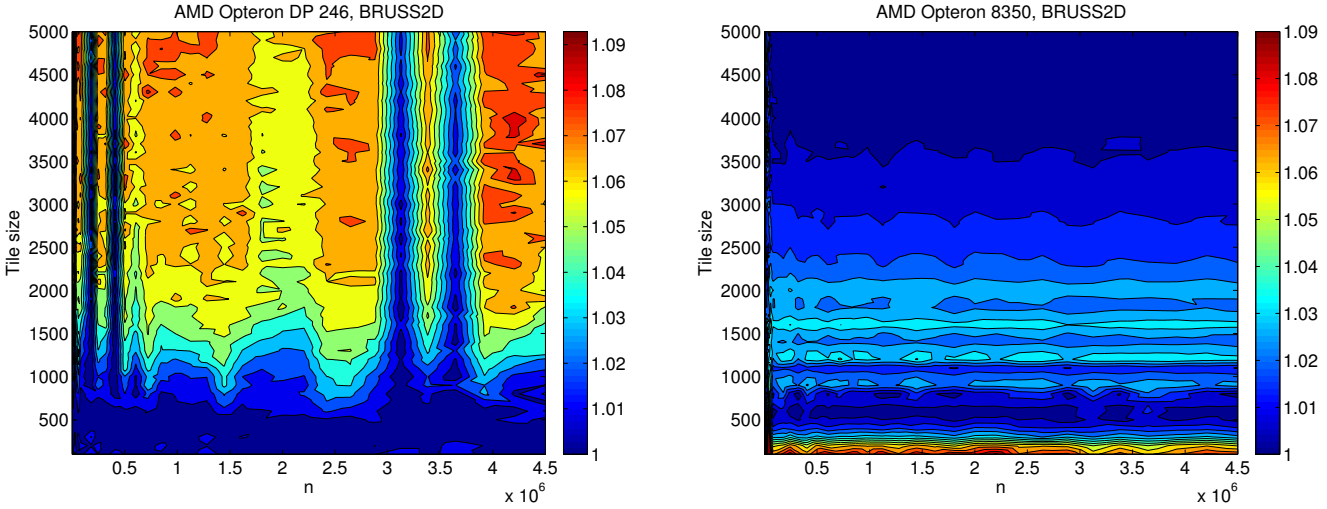

Figure 2: Sensitivity of the execution time of PipeDb2mt to system size and tile size for BRUSS2D and Radau IA (5).

is nevertheless required for highest performance. However, since there usually does not exist a single tile size that is optimal, it is sufficient to select a tile size that lies in a range of tile sizes with acceptable near-optimal performance.

The next subsection describes how such an auto-tuning approach can be integrated into the time-stepping procedure of IRK methods.

5.2 Time-Stepping Phases of the Self-Adaptive IRK Solver

In our work, the self-adaptive IRK algorithm exploits the time-stepping nature of ODE solvers to find the best suitable implementation variant dynamically at runtime. The time-stepping procedure of the self-adaptive IRK algorithm, which can easily be generalized to other time-stepping algorithms, consists of four phases:

1. The pre-selection phase is executed before performing any time steps. In the pre-selection phase, specialized implementation variants are excluded from the candidate pool that are not applicable to the ODE system to be solved in the current run of the solver. Currently, all specialized implementations in our candidate pool are applicable if the ODE system has a limited access distance. The pre-selection phase checks if the ODE system has a limited access distance. If so, the specialized implementations are kept in the candidate pool. Otherwise, the specialized implementations are excluded from the candidate pool.

2. In the second phase of the procedure, which is also executed before the first time step, the algorithm determines a set of appropriate tile size samples for each implementation variant in the candidate pool.

3. After the selection of a set of tile size samples, the algorithm proceeds with the auto-tuning phase, which takes place during the first time steps of the integration. In this phase, the algorithm successively executes all implementation variants with the previously determined corresponding set of tile size samples and determines the fastest implementation variant.

```
1: let P be the problem instance to be solved;
2: let C be the candidate set of implementation variants
3: // pre-selection phase
4: C_1 ← pre-select (C, P);          // problem-based pre-selection
5: // select tile size samples
6: B ← select_tile_size_samples(C_1);
7: // auto-tuning phase
8: t ← t_0;                  // set t to start of integration interval
9: h ← h_init;                                  // select initial step size
10: T_best ← ∞;          // runtime of best implementation so far
11: select an (arbitrary) untiled implementation c ∈ C_1;
12: compute_time_step(P, c, n, t);                    // warm-up step
13: stepsize control; advance t if error small enough;
14: while (C_1 ≠ ∅)
15: {
16:     select an (arbitrary) implementation c ∈ C_1;
17:     C ← C \ c;
18:     BS ← set_of_tile_sizes(B, c);
19:     while (BS ≠ ∅ ∧ t < t_e)
20:     {
21:         select bs ∈ BS;
22:         BS ← BS \ bs;
23:         T ← compute_time_step(P, c, bs, t);
24:         if (T < T_best)    // implementation c is fastest so far
25:         {
26:             c_best ← c;
27:             T_best ← T;
28:             bs_best ← bs;
29:         }
30:         stepsize control; advance t if error small enough;
31:     }
32: }
33: // all implementation variants have been evaluated
34: while (t < t_e)        // remaining part of integration interval
35: {
36:     compute_time_step(P, c_best, bs_best, t);
37:     stepsize control; advance t if error small enough;
38: }
```

Figure 3: Self-adaptive time-stepping procedure.

4. In the last phase, the algorithm uses the fastest implementation variant to compute all remaining time steps.

The pseudo-code notation of the self-adaptive time-stepping procedure is given in Figure 3.

5.3 Selection of Tile Size Samples

Finding the optimal size for a tile is a non-trivial problem. The search space of possible tile sizes is tremendous, which makes an exhaustive search impracticable. In our work, we attempt to reduce the search space of potential tile sizes by a model-based determination of a set of appropriate tile size samples.

The goal of tiling is to maximize the reuse of data elements and to minimize the number of accesses to the relatively slow main memory. The tile size should be selected such that the working space of a tile fits in the cache. Due to the complexity of modern cache hierarchies, it is, however, not obvious for which cache level the tile size should be optimized. But often, at least near-optimal performance can be achieved by optimizing for the fastest level of the memory hierarchy.

The set of appropriate tile size samples is generated for each implementation variant that uses tiling. Let us define the working space of a program segment as the amount of data referenced in this program segment. As a first step towards the selection of appropriate tile sizes, we identify significant working spaces for each of the implementation variants. The largest and most important working space, which contains all data accessed during the execution of the IRK solver, is the working space of one time step (TS). Another

PipeDb2mt	
TS	$(2s+3)n + ts$
CS	$(2s+1)n + ts$
WS_1	$(2s+3)ts + s \cdot 2d(\mathbf{f})$
WS_2	$(s+4)ts + 2d(\mathbf{f})$
WS_3	$2 \cdot ts$
ppDb1mt	
TS	$(s(2m-2)+1)ts + (s+3)n$
PS	$((3s+1)m+4)ts$
WS_1	$(2s+3)ts + s \cdot 2d(\mathbf{f})$
WS_2	$(2s+2)ts + s \cdot 2d(\mathbf{f})$
WS_3	$(s+3)ts + 2d(\mathbf{f})$
WS_4	$2 \cdot ts$

Table 2: Selected working spaces for the implementation variants PipeDb2mt and ppDb1mt. ts is the tile size, n is the size of the ODE system, s is the number of stages, m is the number of corrector steps, and $d(\mathbf{f})$ is the access distance of the ODE system.

important working space for the implementation variants PipeDb2mt, PipeDb2m, Dblock, PipeDb1mt PipeDb1mt, PipeDb1m is the working space of one corrector step (CS), and for the implementation variants ppDb1mt and ppDb1m the working space of one pipelining step (PS). In addition to the working spaces mentioned above, we select at most four other significant working spaces (WS_1, WS_2, WS_3, WS_4) for each of the implementation variants. As examples, Table 2 shows the selected working spaces for the general implementation variant PipeDb2mt and the specialized implementation variant ppDb1mt.

For tile size selection, we take into account the size of the ODE system (n), the (reduced) sizes of the j cache levels of the given architecture denoted as $C(\text{L1}), C(\text{L2}), \ldots, C(\text{L}j)$, and the cache line size of the L1 cache (LS). The unit for the cache sizes and the cache line size is the number of double precision values that can be stored. Since the working spaces described in Table 2 include only vector elements, we consider scalar variables, the coefficients of the RK method used as base method, and operating system data that may be located in cache by reducing the physical cache sizes $C_{\text{phys}}(\text{L}i)$, $i = 1, \ldots, j$, by a safety factor $f_s = 0.9$ such that $C(\text{L}i) = f_s \cdot C_{\text{phys}}(\text{L}i)$. Thus, by using only up to 90 % of the physical cache sizes for the computed working spaces, we try to make sure that the amount of data actually needed by a loop during program execution will not exceed the size of the cache level for which the loop is optimized.

In order to limit the search space, i.e., the number of possible tile sizes to be sampled at runtime, we consider only a small set of tile sizes that make important working spaces likely to fit in the cache hierarchy. Our strategy for the selection of tile size samples is to keep the data of each of the important working spaces in the fastest cache level it fits in. The strategy we pursue by constructing the set of tile size samples can be summarized as follows:

1. Identify the set $\mathcal{W}$ consisting of significant working spaces for the implementation variant considered.

2. For each working space $I \in \mathcal{W}$ and each cache level i compute $LT(I, C(\text{L}i))$, which is the largest tile size for

1: let $\mathcal{W} = \{TS, CS, WS_1, WS_2, \ldots WS_i\}$ be the set of significant working spaces and $\mathcal{C} = \{\mathrm{L1}, \mathrm{L2}, \ldots \mathrm{L}j\}$ the set of cache levels
2: let $\mathcal{BS}$ be the set of tile size samples to be computed
3: $ts \leftarrow \min\{LT(I, C(L)) \mid I \in \mathcal{W}, L \in \mathcal{C}\}$;
4: $\mathcal{BS} \leftarrow \{ts\}$;
5: **if** $(ts \geq 16 \cdot LS + 100)$ $\mathcal{BS} \leftarrow \mathcal{BS} \cup \{16 \cdot LS\}$;

Figure 4: Computation of tile size samples for general implementation variants with loop tiling.

which working space I fits in cache level i. If working space I does not fit in cache level i, $LT(I, C(\mathrm{L}i))$ is set to the size of the ODE system, n.

$$LT(I, C(\mathrm{L}i)) = \begin{cases} \max\{ts \in \{1, \ldots, \min(n, C(\mathrm{L}i))\} \mid \\ \qquad \text{ws\_size}(I, ts) \leq C(\mathrm{L}i)\}, \\ \quad \text{if } \exists\, ts : \text{ws\_size}(I, ts) \leq C(\mathrm{L}i), \\ n, \qquad \text{otherwise.} \end{cases} \tag{5}$$

The utility function $\text{ws\_size}(I, ts)$ computes the size of working space I for tile size ts and is defined as follows:

$$\text{ws\_size}(I, ts) = \begin{cases} \text{size of } I \text{ for tile size } ts, \\ \quad \text{if } ts \in \{1, \ldots, n\}, \\ n, \qquad \text{otherwise.} \end{cases} \tag{6}$$

3. Select the tile size ts as the minimum of all tile sizes computed in the previous step. Even though it may not be possible to fit all working spaces in the L1 cache, this tile size selection strategy guarantees that the working spaces considered stay in the fastest cache level they individually fit in.

4. In addition, include the tile size $16 \cdot LS$ in the set of tile size samples.

Figure 4 describes the procedure used to select tile size samples for general implementation variants that use loop tiling. In line 5, we perform an additonal check if ts satisfies the heuristic property $ts \geq 16 \cdot LS + 100$. If so, a small tile size of length $16 \cdot LS$ is included in the set of tile size samples, because we have observed that on some processors, in particular, on the AMD Opteron DP 246, such a small tile size is sometimes more effective than a tile size for which the optimized working spaces occupy a large part of the cache.

To select the tile size samples for the specialized implementation variants, we pursue a strategy similar to the one described in Figure 4. When selecting the tile size samples for the specialized implementation variants, additional constraints have to be satisfied. The first constraint is $ts \geq d(\mathbf{f})$, i.e., we have to guarantee that the selected tile size is larger than the ODE access distance. The second constraint, which is required only for the pipelining variants (ppDb1mt and ppDb1m), is $ts \leq \lfloor n/m \rfloor$. For the specialized implementation variants, the heuristic property in line 5 is replaced by the check if $d(\mathbf{f}) \geq 16 \cdot LS$. If this property is true, $d(\mathbf{f})$, otherwise $16 \cdot LS$ is included in the set of tile size samples. For both specialized and general implementation variants, the set of tile size samples consists of at most two tile sizes for each of the implementation variants.

5.4 Implementation of the Auto-tuning Phase

The implementation variants in the candidate pool differ in the order of the computations performed, but since all implementation variants in the candidate pool produce the same numerical results, they can be exchanged for the computations of the time-steps.

In the auto-tuning phase, the algorithm successively executes all available implementation variants in the candidate pool. For each implementation variant, a runtime test is performed with different tile sizes to select empirically the best tile size from the set of tile size samples. The time needed by each implementation variant for each tile size to compute one time step is measured, and the fastest implementation variant and the best tile size are recorded. Because the time needed to compute one time step and, in particular, the differences in the execution times of one time step of the different implementation variants may be smaller than the resolution of portable Unix timer functions, we evaluate hardware performance counters to count CPU cycles. The PAPI (Performance Application Programming Interface) library [15] provides a portable way to access these counters.

Generally, the performance of programs depends on the initial state of the hardware system, in particular, the contents of the cache hierarchy. It is, therefore, usually necessary to start the evaluation of a program from a reproducible initial state of the cache hierarchy, which is called cache warm-up. In order to warm up the cache reproducibly and thus to assess the implementation variants as reliably as possible, the self-adaptive IRK algorithm presented in [11] distinguishes between accepted and rejected steps, because an additional vector copy is required to discard the approximation vector $\mathbf{y}_{\kappa+1}$ computed in the current time step when the time step is rejected. Since usually only a small part of the time steps is rejected, accepted time steps are used to measure the runtime of the implementation variants. But, to warm-up the cache reproducibly, only those accepted steps are considered, which directly follow another accepted step.

However, warm-up steps can imply additional overhead, because they extend the auto-tuning phase and shorten the fourth phase in which the best implementation variant is used. We therefore investigated the impact of warm-up steps on the reliable assessment of the implementation variants. An experimental evaluation has shown that it is sufficient to use the first time step as a single warm-up step to load data used by all implementation variants into the cache. Skipping this first single warm-up step may result in a runtime difference of $\gtrsim 2\,\%$, which may be larger than the performance difference of some implementation variants. For the rest of the time steps in the auto-tuning phase, we observed only a very small difference between warmed-up and not warmed-up time steps of the same implementation variant. Because of this observation, the new algorithm described in this paper reduces the number of steps in the auto-tuning phase by using all time steps regardless of whether they are accepted or have to be rejected to assess the implementation variants. Only the first time step is still used as a warm-up step.

The number of time steps executed in the auto-tuning phase depends on the number of implementation variants kept in the candidate pool and on the number of selected tile size samples. For ODE systems with limited access distance, the auto-tuning phase takes at most 18 time steps. The maximum number of time steps constituting the auto-tuning phase for ODE systems with arbitrary access distance is 8.

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup

In this section, we present an experimental evaluation of the self-adaptive IRK solver with tile size sampling on two different hardware platforms. The first system is an AMD Opteron DP 246 machine running at 2.0 GHz with 64 kB L1 cache and 1 MB L2 cache. The second system is an AMD Opteron 8350 with four quad-core processors running at 2.0 GHz and equipped with 64 kB L1 cache/core, 512 kB L2 cache/core and 2 MB shared L3 cache. The compiler used on both systems was GCC 4.4.3 with optimization level 2.

As example problems we selected the 2D Brusselator equation (BRUSS2D) [8] and a model for a nerve impulse mechanism (CUSP) [9]. Both problems were derived from PDE systems by a spatial discretization using the method of lines. BRUSS2D is discretized on an $N \times N$ grid. The resulting ODE system has size $n = 2N^2$ and a limited access distance of $d(\mathbf{f}) = 2N$. It can be solved using specialized or general implementation variants. CUSP is discretized using a one-dimensional grid consisting of N points resulting in an ODE system of size $n = 3N$. Due to periodic boundary conditions, CUSP has an unlimited access distance. Therefore, the specialized implementation variants currently in the candidate pool cannot be applied to this problem. As corrector method, we used the 5-stage method Lobatto IIIC (8) [8].

The metric used in the experiments to compare the implementation variants is execution time normalized by the number of time steps and the system dimension n. In order to reduce the time required for the experiments with the self-adaptive IRK solvers, we limit the number of time steps such that at least about three times as many time steps are executed as required by the auto-tuning phase and such that the execution time of a program run is still large enough, i.e., about 30 s, so that it can be measured reliably. As a result, the number of time steps executed with the self-adaptive solver varies between about 60 and about 850.

6.2 Applicability of Self-Adaptive Solvers

At first we demonstrate, using BRUSS2D as example, that the general approach followed by the self-adaptive IRK solver is applicable to the two test problems on the two hardware architectures and that this approach enables a good performance with only small overhead.

Figure 5 shows a detailed comparison of the normalized execution times of the previous variant of the self-adaptive IRK solver, as presented in [11], with the non-adaptive implementation variants from the candidate pool. In Figure 5, the normalized execution time is plotted against the system size, n.

When we compare the different implementation variants in Figure 5 for the AMD Opteron 8350 machine, we notice that for different sizes n of the ODE system the order of the implementation variants varies. We can identify subranges where different implementation variants offer the best execution time. For system sizes $n \lesssim 1.6 \cdot 10^4$, the fastest implementation variant is A, then, for $1.6 \cdot 10^4 \lesssim n \lesssim 1.0 \cdot 10^6$, ppDb1mt is the fastest implementation variant. For system sizes $1.0 \cdot 10^6 \lesssim n \lesssim 2.2 \cdot 10^6$, PipeDb1m delivers the best performance. In the range $2.2 \cdot 10^6 \lesssim n \lesssim 3.7 \cdot 10^6$, PipeDb2mt is the fastest implementation variant. For even larger system sizes, the fastest implementation variant is ppDb1mt again.

On the AMD Opteron DP 246, similary to the AMD Opteron 8350 machine, we can identify sub-ranges, where the order of the implementation variants delivering the best performance changes. In comparison to the non-adaptive implementation variants, the self-adaptive IRK solver achieves nearly the same performance as the best non-adaptive variant in each of the sub-ranges on both machines.

The time overhead caused by the auto-tuning phase is acceptably small, even though the number of time steps executed in our experiments was limited to a few hundred steps to reduce the time required to conduct the experiments. In real simulation runs, which may require millions of time steps, the overhead of the auto-tuning phase would be negligible.

6.3 Impact of Tile Size and Warm-Up

To investigate the impact of the runtime tile size selection and warm-up steps we compare three different variants of the self-adaptive IRK solver:

1. The previous variant of the self-adaptive IRK algorithm, (A^{prev}), presented in [11]. This variant does not carry out any procedure for the selection of tile sizes. Instead, it uses the same predefined tile size for all implementation variants in the candidate pool.

   For BRUSS2D, the access distance $d(\mathbf{f})$ is used as tile size, because for BRUSS2D the specialized implementation variants are applicable and the access distance is the smallest tile size that can be selected for the specialized variants. The access distance also is usually large enough to exploit spatial and temporal locality efficiently. For ODE problems with unlimited access distance, e.g., CUSP, the specialized implementation variants are not applicable and the tile size is set to 120, which corresponds to 15 cache lines on the machines considered in our experiments, which use a typical cache line size of 64 byte. The tile size 120 is chosen, because previous experience has shown that this value leads to good results on different machines.

   To warm-up the cache, only those accepted steps are considered for assessing the implementation variants, which directly follow another accepted step.

2. The second variant, (A^{ts}), is the self-adaptive variant of the IRK method, as shown in Figure 3, which performs dynamic selection of tile sizes. It uses only the first time step of the integration to warm-up the cache.

3. The third self-adaptive variant, (A^{nts}), is similar to (A^{ts}), but does not perform tile size sampling. The tile sizes are predefined as in (A^{prev}).

The normalized execution times of these three variants of the self-adaptive IRK solver and the tile sizes selected are shown in Figure 6.

6.3.1 Execution Times on AMD Opteron 8350

For BRUSS2D, the two self-adaptive solvers (A^{prev}) and (A^{nts}), which both do not perform tile size selection, always obtain a similar performance and usually select the same implementation variant. Hence, it is not necessary to perform warm-up steps for each implementation variant, but also no significant gain from omitting the warm-up steps could be observed on this machine.

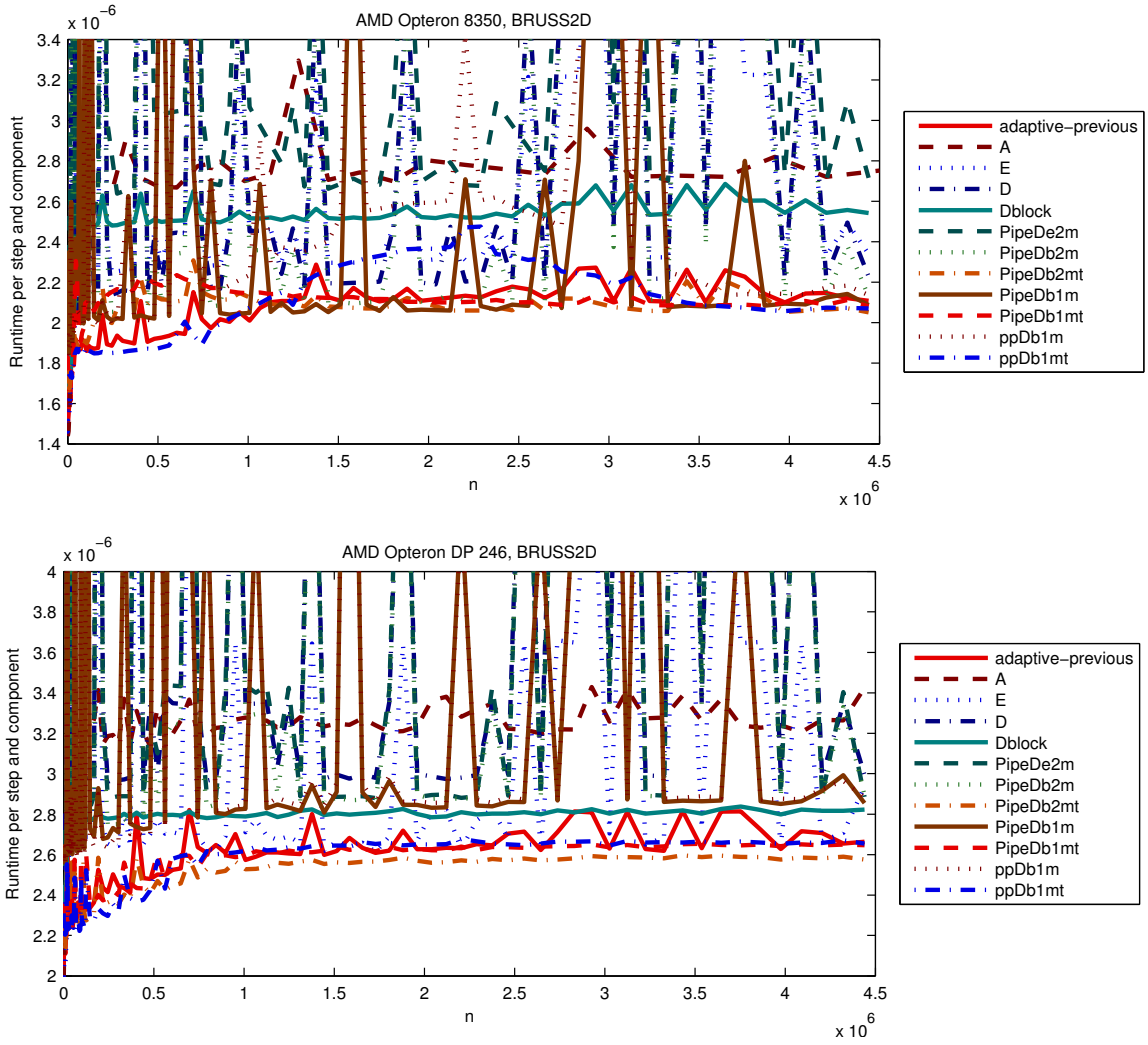

Figure 5: Normalized execution times for varying system sizes of BRUSS2D for Lobatto IIIC (8).

For system sizes $n \lesssim 1.0 \cdot 10^6$, all self-adaptive implementation variants obtain a very similar performance. In this range of n, either untiled general implementation variants or ppDb1mt obtain the best performance. Even though ppDb1mt uses loop tiling, the performance of this implementation cannot be improved by tile size selection for BRUSS2D in this range. The reason is that the specialized implementation variants such as ppDb1mt subdivide the ODE system into blocks that have to be larger than the access distance $d(\mathbf{f})$, where the size of the blocks is equal to the tile size ts. This constraint limits the choice of possible tile sizes, since tile sizes smaller than the access distance cannot be selected. However, larger tile sizes than the access distance $d(\mathbf{f})$ do not improve the performance on this machine.

For BRUSS2D with system sizes $n \gtrsim 1.0 \cdot 10^6$, the situation changes and the self-adaptive variant (A^{ts}), which performs tile size selection, offers the best performance. In this range of n, ppDb1mt no longer obtains the best performance, because the working space of one pipelining step, which grows with n, because $d(\mathbf{f}) = 2N = 2\sqrt{n/2}$ also grows, becomes too large. Therefore, the performance of ppDb1mt for $ts = d(\mathbf{f})$ becomes very similar to the performance of PipeDb1m and PipeDb2mt, and the self-adaptive solvers that do not perform tile size selection, choose one or another of these three implementations. The self-adaptive solver (A^{ts}), however, chooses PipeDb2mt in most runs with either a smaller or higher tile size than $d(\mathbf{f})$, which leads to a higher performance.

6.3.2 *Execution Times on AMD Opteron DP 246*

For this machine, we can make similar observations as for the AMD Opteron 8350 machine. Again, there is no large difference in the performance of (A^{prev}) and (A^{nts}) for BRUSS2D. Thus, the impact of warm-up steps is only small. However, the range in which (A^{nts}), which does not perform warm-up steps, is noticeably faster then (A^{prev}), which warms-up each implementation, is larger.

As for the AMD Opteron 8350 machine, (A^{ts}) outperforms the variants without tile size selection when the working space of one pipelining step of ppDb1mt grows too large. We observe that for $n \gtrsim 2.3 \cdot 10^5$ PipeDb2mt with a small tile size is faster than ppDb1mt. The performance improvement

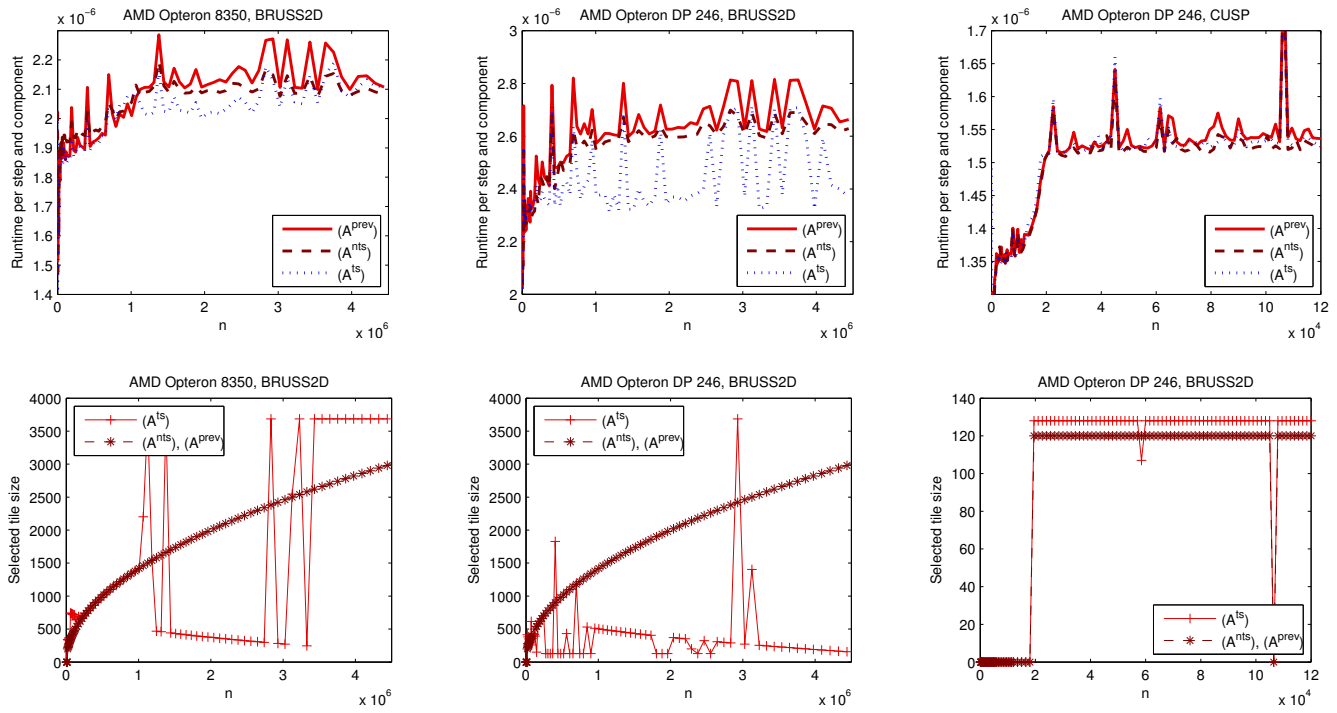

Figure 6: Normalized execution times and tile sizes selected for the different variants of the self-adaptive IRK solver for varying system sizes.

resulting from selecting PipeDb2mt with a small tile size is significantly larger than on the AMD Opteron 8350, because this machine is more sensitive to the choice of the tile size (cf. Figure 7).

For CUSP, only the general implementation variants could be included in the candidate pool. The performance of all three self-adaptive variants is very similar over the entire range of n considered. The reason is that on this machine small tile sizes are preferable. Hence, for small system sizes $n \lesssim 2 \cdot 10^4$, where the working space of one time step fits in the cache, all self-adaptive solvers select the untiled implementation variant E, while, for larger system sizes, all self-adaptive solvers select the tiled implementation variant PipeDb2mt. The tile size chosen for PipeDb2mt by (A^{ts}) in most runs is $128 = 16 \cdot LS$. This value nearly matches the predefined tile size of 120 used by the self-adaptive solvers without tile size selection.

6.3.3 Quality of the Tile Size Samples

For BRUSS2D as example problem and Lobatto IIIC (8) as base method, Figures 7 compares the quality of the tile size samples ts_1 and ts_2 generated by the self-adaptive algorithm for implementation variant PipeDb2mt with the quality of the fixed tile size $d(\mathbf{f})$ used for the non-adaptive execution of implementation variant PipeDb2mt on the AMD Opteron DP 246 machine and on the AMD Opteron 8350 machine. In the contour plots, the relative runtimes for implementation variant PipeDb2mt are plotted w.r.t. the best runtime over the range of tile sizes for different system sizes n. The color scale indicates the quality of the selected tile sizes. The deviation in the performance for the range of tile sizes of $[1, 5000]$ considered is $\lesssim 13\,\%$ on the AMD Opteron DP 246, $\lesssim 11\,\%$ on the AMD Opteron 8350.

On both machines we observe that the self-adaptive algorithm generates tile size samples that mainly lie in blue regions of the contour plots, which are associated with a good choice for the tile size. The tile sizes used for different n for the non-adaptive execution of PipeDb2mt mostly fall into red regions corresponding to ranges with less suitable tile sizes. On the AMD Opteron DP 246 machine, we observe that the tile size selected should not be larger than ≈ 700 over the entire range of n. In contrast, on the AMD Opteron 8350, not only small tile sizes in the range of $[200, 500]$ show a good performance, but also larger tile sizes in the range of $[3500, 5000]$. On both machines, the self-adaptive algorithm generates the same tile sizes, because both machines are equipped with 64 kB L1 cache. For machines with the same L1 cache size, the self-adaptive algorithm proposes the same tile size samples as long as there exists a workings space that fits in the L1 cache with a computed tile size that is smaller than all tile sizes computed for other working spaces fitting only in the L2 or the L3 cache.

6.3.4 Correlation with Cache Miss Counts

Figure 8 shows the normalized L1 and L2 cache miss counts for varying system sizes and tile sizes on the AMD Opteron DP 246 for implementation variant PipeDb2mt with BRUSS2D as example problem and Lobatto IIIC (8) as base method. In the contour plots, the normalized cache miss counts are plotted w.r.t. the minimum number of normalized cache misses over the range of tile sizes for different system sizes n. We see a sharp increase in the number of L1 misses when the tile size grows, whereas the number of L2 cache misses does not increase for larger tile sizes and only a small variation of the number of L2 cache misses for different tile sizes and system sizes is observable. The vari-

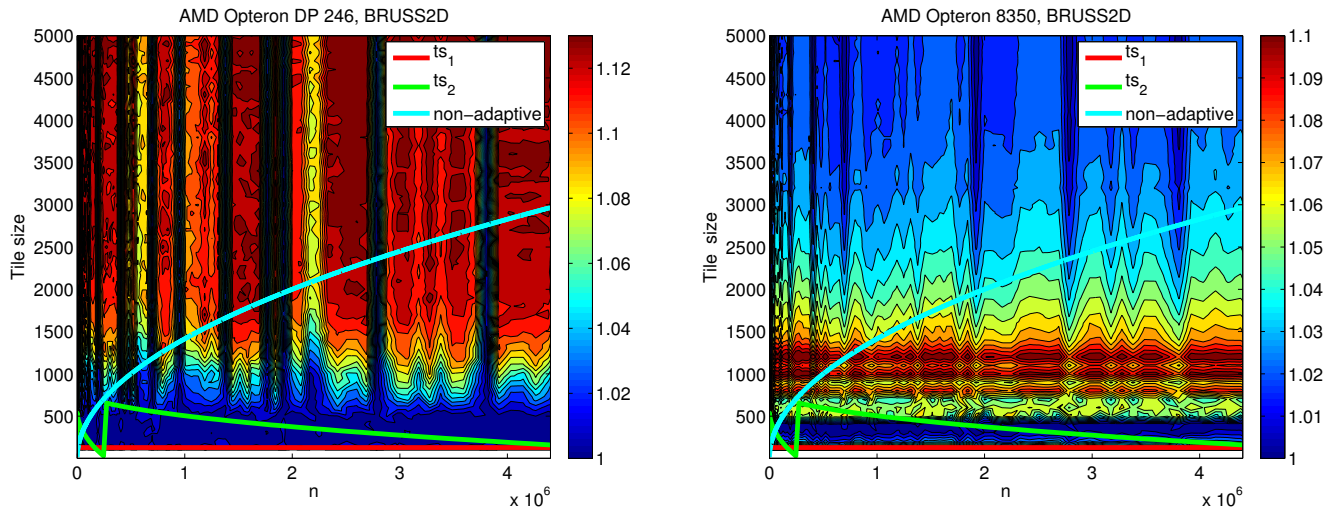

Figure 7: Quality of the tile size samples for implementation variant PipeDb2mt with BRUSS2D as example problem and Lobatto IIIC (8) as base method.

ation of the L1 cache misses for different tile sizes is higher than the variation of the L2 cache misses by several orders of magnitude.

With growing tile size and system size, the sizes of several working spaces exceed the capacity of the L1 cache and, consequently, the number of L1 cache misses increases. Since most of the working spaces still fit in the L2 cache with a tile size $\lesssim 5000$, the number of L2 cache misses remains almost constant. From the runtimes measured and the numbers of cache misses shown in Figure 8, we can conclude that cache capacities and working space sizes are the major factors that need to be considered for tile size selection. In order to obtain a good performance for an implementation variant, the tile size should be selected such that the data of each of the important working spaces are kept in the fastest cache level the corresponding working space fits in.

7. CONCLUSIONS

In this paper, we have investigated dynamic auto-tuning techniques for sequential IRK methods. We have compared several variants of a self-adaptive IRK solver, which exploits the time-stepping nature of the solution procedure to select the best implementation variant from a candidate pool at runtime. In particular, we have investigated the empirical selection of tile sizes for loop tiling from a set of tile size samples, where the set of tile size samples is determined by a model-based approach.

A detailed experimental evaluation has shown that the self-adaptive IRK solver can be applied successfully to solve different IVPs on different hardware architectures efficiently because the overhead required for the dynamic selection of the best implementation variant and a suitable tile size is sufficiently small. These results are very encouraging, and we still see potential for improvements of the self-adaptive algorithm. In particular, our experiments have shown that it is important to further reduce the candidate set of implementation variants and tile sizes.

IRK methods were selected for this study because of their high potential for parallelism. As a next step of our work, we will consider shared-memory implementations of IRK methods, whose locality behavior and performance additionally depends on the number of processors used.

8. ACKNOWLEDGMENTS

This work has been supported by the German Research Foundation (DFG).

9. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2nd edition, 2007.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence Based Approach*. Morgan Kaufmann, 2002.

[3] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *11th ACM International Conference on Supercomputing (ICS'97)*, 1997.

[4] K. Burrage. *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford University Press, New York, 1995.

[5] R. Ehrig, U. Nowak, and P. Deuflhard. Massively parallel linearly-implicit extrapolation algorithms as a powerful tool in process simulation. In *Parallel Computing: Fundamentals, Applications and New Directions*, pages 517–524. Elsevier, 1998.

[6] V. Eijkhout and E. Fuentes. *New Advances in Machine Learning*, chapter Machine Learning for Multi-stage Selection of Numerical Methods. INTECH, Feb. 2010.

[7] M. Frigo, Steven, and G. Johnson. The design and implementation of FFTW3. In *Proceedings of the IEEE*, pages 216–231, 2005.

[8] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, Berlin, 2nd rev. edition, 2000.

[9] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and*

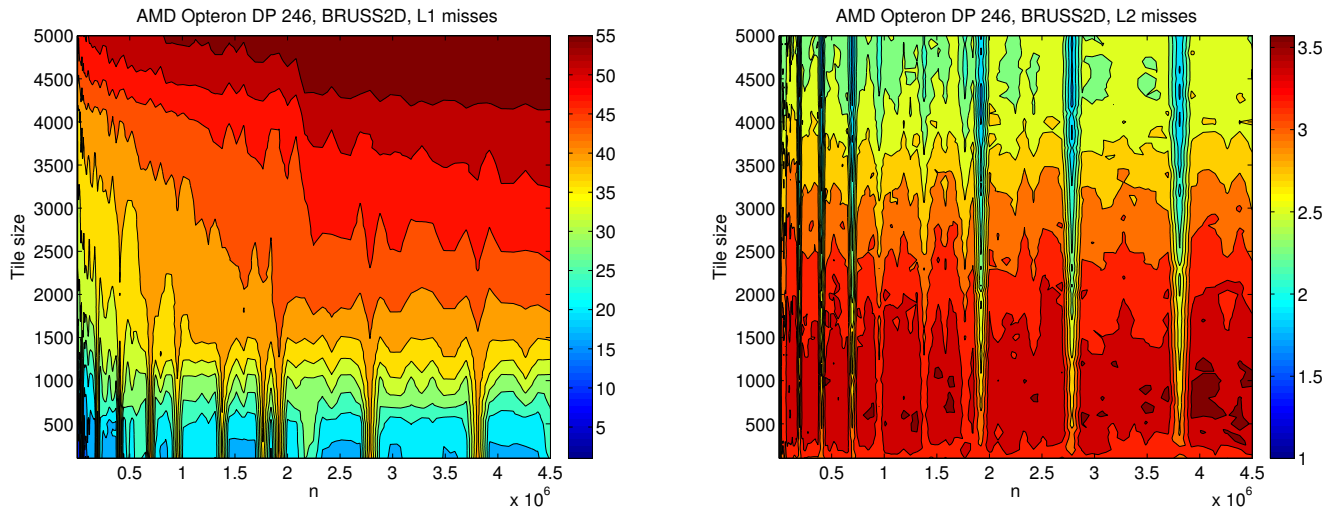

Figure 8: Normalized number of L1 and L2 cache misses for varying system sizes and tile sizes for PipeDb2mt with BRUSS2D as example problem and Lobatto IIIC (8) as base method on AMD Opteron DP 246.

Differential-Algebraic Problems. Springer, Berlin, 2nd rev. edition, 2002.

[10] A. Hartono, M. Baskaran, J. Ramanujam, and P. Sadayappan. DynTile: Parametric tiled loop generation for parallel execution on multicore processors. In *IPDPS 2010: Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing*. IEEE Computer Society, Apr. 2010.

[11] N. Kalinnik, M. Korch, and T. Rauber. Applicability of dynamic selection of implementation variants of sequential iterated Runge-Kutta methods. In *2010 IEEE International Conference on Cluster Computing – Workshops and Tutorials*. IEEE Computer Society, 2010.

[12] M. Kappeller, M. Kiehl, M. Perzl, and M. Lenke. Optimized extrapolation methods for parallel solution of IVPs on different computer architectures. *Applied Mathematics and Computation*, 77(2–3):301–315, July 1996.

[13] M. Korch and T. Rauber. Locality optimized shared-memory implementations of iterated Runge-Kutta methods. In *Euro-Par 2007. Parallel Processing*, volume 4641 of *LNCS*, pages 737–747. Springer, 2007.

[14] S. P. Nørsett and H. H. Simonsen. Aspects of parallel Runge–Kutta methods. In *Numerical Methods for Ordinary Differential Equations*, number 1386 in LNM, pages 103–117, 1989.

[15] Performance Application Programming Interface (PAPI) homepage. `http://icl.cs.utk.edu/papi/`.

[16] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*, Nov. 2010.

[17] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[18] M. Rahman, L.-N. Pouchet, and P. Sadayappan. Neural network assisted tile size selection. In *International Workshop on Automatic Performance Tuning (IWAPT'2010)*, Berkeley, CA, June 2010. Springer.

[19] B. A. Schmitt, R. Weiner, and S. Jebens. Parameter optimization for explicit parallel peer two-step methods. *Appl. Numer. Math.*, 59:769–782, 2008.

[20] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE Computer Society, 2009.

[21] P. J. van der Houwen and B. P. Sommeijer. Parallel iteration of high-order Runge–Kutta methods with stepsize control. *J. Comput. Appl. Math.*, 29:111–127, 1990.

[22] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, University of Tennessee, 1997.

[23] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358 –386, Feb. 2005.

[24] J. Zhao, M. Horsnell, M. Luján, I. Rogers, C. Kirkham, and I. Watson. Adaptive loop tiling for a multi-cluster CMP. In *ICA3PP '08: Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, pages 220–232. Springer, 2008.