Modeling the Effect of Application-specific Program Transformations on Energy and Performance Improvements of Parallel ODE Solvers

appeared in: Journal of Computational Science, Bd. 51. Elsevier ISSN 1877-7503, 2021. DOI: 10.1016/j.jocs.2021.101356

Thomas Rauber

Computer Science Department, University Bayreuth, Germany

Gudula Rünger

Computer Science Department, Chemnitz University of Technology, Germany

Abstract

Ordinary differential equations (ODEs) are important for modelling many problems from science and engineering and efficient ODE solvers are required, for example when solving time-dependent partial differential equations (PDEs) with the method of lines. Since an ODE solver may perform a large number of iteration steps, the execution time for solving an ODE problem might be quite large. Thus, a reduction of the execution time is desirable and should affect each iteration step of the simulation. Programming techniques to reduce the execution time of ODE solver are parallelism and modification of the memory access structure such that the memory access time decreases. In this article, we investigate multithreaded solution methods for ODEs with different memory access behavior and their influence on the performance. Additionally the energy consumption is considered. The parallelism is implemented as shared memory program for multicore processors. The memory access behavior is investigated using different program variants which result from application-specific program transformations changing the memory access order while guaranteeing the numerical correctness. For the investigation of the performance, experimental data have been gathered on five different recent multicore processors. Additionally, an analytical power and energy model for modeling the performance and energy consumption is introduced. As ODE solver, the popular embedded Runge-Kutta methods with error correction is used. The simulation problems are two different ODEs resulting from discretized PDEs. The experimental data give insight into the quite diverse performance behavior of the ODE solver variants solving the same problem on different platforms.

Email addresses: `rauber@uni-bayreuth.de` (Thomas Rauber), `ruenger@informatik.tu-chemnitz.de` (Gudula Rünger)

Keywords: systems of ordinary differential equations, solution methods for ODEs, multithreaded ODE solvers, application-specific programs transformations, performance and energy analysis, frequency scaling.

1. Introduction

Power and energy awareness are important aspects for the design of modern processors. Several features, such as frequency scaling or power capping, have been developed and are integrated in power management units (PMUs) to control the power and energy consumption at hardware level. Recent processors have several performance states (P-states) and CPU operating states (C-states) which allow the hardware to react to different workloads or to turn off unused components to save power. Dynamic voltage and frequency scaling (DVFS) is supported by most processors to reduce the energy consumption.

Besides the importance of hardware mechanisms to control the energy consumption, software aspects are also of high interest, since the software behavior may have a large influence on the resulting energy consumption and execution time. Thus, it is essential to design software such that the resulting energy consumption meets the given requirements, e.g. be as small as possible. Program transformations may play an important role in this context, since they may lead to program versions with a different performance, power or energy consumption. A crucial step towards designing energy-aware software using program transformations is the ability to assess the energy and performance effects of specific program transformations. A final goal is to quantitatively understand which transformation has which effect on the resulting energy and performance behavior. In this article, we contribute to this goal in the context of ODE solvers. In particular, we apply program transformations to create new program versions and explore the effects of several application-specific transformations on the performance and energy consumption of the new program versions on several recent multicore processors. The transformations applied are difficult to find by a compiler due to the complex program structure with intertwined loops and function calls. We also show that power and energy models can help to get insight into the observed performance and energy behavior.

The solution of ordinary differential equations (ODEs) is an important area in scientific computing and, thus, their performance and energy consumption is of large interest. In this article, we investigate embedded Runge-Kutta (RK) methods, which are popular ODE solvers for a broad range of application problems, including discretized time-dependent PDE problems, often resulting in large systems of ODEs [20]. Since embedded RK methods are one-step methods possibly performing a large number of iteration steps with function evaluations of the right-hand sinde function of the ODE system, it is crucial to design each iteration step such that it is efficient in terms of execution time and energy consumption.

To provide a suitable basis for the investigation of the effect of program transformations, we have developed several implementation versions for embedded RK methods resulting from the usage of a series of consecutive application-specific program transformations. The final RK version implements an RK method with delayed function evaluations of the right-hand side of the ODE system. For each solver version, we have developed a multithreaded implementation which exploit the size of the ODE system for an execution on multicore processors in each iteration. However, synchronization is needed between iterations due to data dependencies. The interaction of the multithreaded implementations and the loop transformations lead to further requirements for synchronization points within the individual iterations, which are necessary to guarantee the numerical correctness. This issue is discussed together with the program transformations.

The contributions of this work are in the area of application-specific program transformations and the investigation of their effect on the resulting performance, power, and energy consumption of ODE solvers for five different multicore processors. Several influencing factors are analyzed in detail:

- The effect of the computational demands and the memory access characteristics of the specific ODE problem to be solved: Two ODE problems with different characteristics are considered and used as test cases and an experimental evaluation on different desktop and server processors is performed.
- The effect of the number of threads used to solve the ODE problems: It is shown that the program transformations may have different effects for a varying number of threads. The resulting performance scalability and energy consumption strongly depends on the computational characteristics of the specific ODE problem to be solved.
- The effect of frequency scaling using DVFS on the resulting performance: For different operational frequencies, different performance, power, and energy effects can be observed for the ODE solver versions resulting from the program transformations.
- The effect of the processor architecture: The experimental evaluation is performed on five multicore systems with different numbers of cores. It is shown that different multicore systems may have quite different power consumption, resulting in large differences in the overall energy consumption.

The rest of the paper is structured as follows. Section 2 describes the different multithreaded implementation versions of ODE solution methods. Section 3 introduces the power and energy model used for DVFS. Section 4 contains the experimental evaluation on different hardware systems. Section 5 uses the models from Section 3 for a modeling of execution time and energy consumption. Section 6 discusses related work. Section 7 gives some concluding remarks.

2. Solution methods for ODEs

The program transformations are applied to the explicit RK method, which is briefly summarized in Subsect. 2.1. The series of transformations is described in 2.2 and the resulting multithreaded implementations are given in Subsect. 2.3.

2.1. Explicit RK methods

An initial value problems for systems of ODEs of size $n \geq 1$ is given by:

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

where $\mathbf{y}_0 \in \mathbb{R}^n$ at start time x_0 is an initial vector and $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ is the right hand side function. Explicit RK methods are one-step methods executing a series of consecutive time steps $\kappa = 0, 1, 2, \ldots$ in which new approximation vectors $\eta_{\kappa+1} \in \mathbb{R}^n$ for the unknown solution function $\mathbf{y}(x_{\kappa+1})$ at position $x_{\kappa+1}$ are computed. For the computation of one approximation vector $\eta_{\kappa+1}$, an s-stage RK method uses s stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s \in \mathbb{R}^n$, which are computed according to:

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

The number s of stage vectors is fixed for the specific method. The computation of the next approximation vector $\eta_{\kappa+1}$ from the previous approximation vector $\eta_{\kappa+}$ includes the computation of an additional approximation vector $\hat{\eta}_{\kappa+1}$ of lower order, which is used for error control and stepsize adaption:

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

The s–dimensional vectors $b = (b_1, \ldots, b_s)$, $\hat{b} = (\hat{b}_1, \ldots, \hat{b}_s)$ and $c = (c_1, \ldots, c_s)$ and the $s \times s$ matrix $A = (a_{il})$ are specific for a particular RK method. The order r of the approximation $\eta_{\kappa+1}$ and the order $\hat{r}$ of the approximation $\hat{\eta}_{\kappa+1}$ usually differ by 1, i.e. $r = \hat{r} + 1$. The difference between the two approximations $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ provides an asymptotic estimate of the local error in the lower order approximation and is used for stepsize control [12]. The approximation of the current time step is accepted, if a suitable weighted norm of the local error estimate lies within a predefined tolerance level. Several embedded RK methods have been proposed in the past. RK methods that are often used include the methods of Dormand & Prince (e.g. DOPRI5 of order $5(4)$ or DOPRI8 of order $8(7)$) and Verner's methods DVERK of order $6(5)$ [20]. As an example for a typical ODE solver, we consider explicit RK methods with an error control and stepsize selection mechanism [20, 12].

An important source for performance improvements of RK methods are global rearrangements of the data accesses to the approximation and stage vectors. However, the computation scheme (2), (3) restricts the potential evaluation order due to data dependencies between the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$, and $\eta_{\kappa+1}$, since all stage vectors have to be computed before the computation of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ can be started. These data dependencies have to be taken into account in order to preserve the numerical correctness. Also, according to the form of the argument vectors of $\mathbf{f}$ in (2), the computation of stage vector $\mathbf{v}_i$ depends on $\mathbf{v}_1, \dots, \mathbf{v}_{i-1}$, $i = 2, \dots, s$, so that the stage vectors have to be computed one after another. For a general ODE solver, the dependence structure of $\mathbf{f}$ of a specific ODE system is not known in advance and therefore the conservative assumption that every component of $\mathbf{f}$ depends on all vector components of its argument vector has to be made for a generally applicable implementation. Hence, the computation of one component of stage vector $\mathbf{v}_i$ requires that all components of $\mathbf{v}_1, \dots, \mathbf{v}_{i-1}$ have already been computed. Since $\mathbf{f}$ may access all these components, i vectors of size n have to fit into the cache simultaneously to avoid capacity misses. The maximum size of data $s \cdot n$ required within one time step is reached when computing the last stage vector $\mathbf{v}_s$. This volume of data is needed for the computation of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ in Equ. (3).

2.2. Application-specific program transformations

The implementation of the RK methods proposed in this article have a generic form as described by computation scheme (2), (3). Thus, the implementation is suitable for arbitrary embedded RK methods, and several RK methods are implemented, including RadauIA, LobattoIIIA, RadauIIA, LobattoIIIC, and DVERK of different order. As hardware platform, we assume a shared address space as provided by multicore processors. Based on the Pthreads library, we have developed several multithreaded implementations that are applicable for arbitrary embedded RK methods. The implementations differ in their loop structure and the synchronizations that are needed to guarantee the correct numerical behavior.

The main potential for a parallel execution by multiple threads is given by the ODE system to be solved. Several arrays of size n capture the essential parameters of the RK scheme (2), which are the stage vectors $\mathbf{v}_1, \dots, \mathbf{v}_s$ and the two approximation vectors η_κ and $\hat{\eta}_\kappa$. All multithreaded implementations exploit this potential of parallelism by a data parallel program structure in which several threads update the stage and approximation vectors in a block-wise fashion. The threads are created at the beginning of the time steps of the RK method and are then used for the computation of separate blocks of the argument vectors, stage vectors, and approximation vectors for all time steps performed.

We have derived a series of multithreaded RK implementations, which are derived from one another by applying application-specific program transformations. In the following, we describe four main program versions and omit intermediate implementation versions resulting from minor transformation steps. The four main RK implementations can be summarized as follows:

1. The first implementation version results by a direct translation of the loop structure in computation scheme (2), (3) into program code. According to the loop structure, the implementation version is named i-l-j, where i loops over the stage vectors, l is used for the argument vectors and j is the loop over the dimension of the vectors. The multithreaded implementations are based on a data-parallel computation of the stage vectors and the approximation vectors. This kind of

parallelism is called parallelism across the system [16]. Since we aim at general non-specialized RK implementations, parallelism across the method is not exploited. The parallel computation of the stage and approximation vectors requires the use of barrier synchronizations within the same time step.

2. Due to the order of computations in scheme (2), the stage vectors have to be computed one after another, since v_i is needed for the argument of the function evaluation of $\mathbf{f}$ producing $\mathbf{v}_{i+1}$, $i = 1, \ldots, s-1$. To be able to change the computation order and to enable transformations in the loop structure, we introduce additional arrays that separately hold the argument vectors for the function evaluations needed to compute $\mathbf{v}_1, \ldots, \mathbf{v}_s$. This transformation enlarges the amount of data structures to be stored and accessed during one time step. However, besides the advantages to enable subsequent transformations, there is the additional advantage that less barrier synchronizations are needed, since the threads can start the computation for the next stage vector earlier without affecting the numerical correctness. This implementation version is called i-l-j-ext in the following.
3. A variation of the preceding implementation i-l-j-ext results by a loop exchange with the dimension loop, which carries the parallel execution. More precisely, the iterative computation of $\sum_{l=1}^{i-1} a_{il}\mathbf{v}_l$, which is part of the argument vectors in Equ. (2), is exchanged with the dimension loop, so that each thread has a different computation order. The number of barrier synchronization needed remains unchanged.
4. The next implementation version is derived from the i-j-l-ext Version by exploiting that the additional vectors decouple the computations and make a loop exchange possible. This computation strategy has the effect that each newly computed value is used immediately after its computation to update all dependent values which leads to an improved temporal locality of the memory references. In the resulting loop nest, the inner loop is exchanged with the dimension loop. Thus, each thread computes its portion in a different order and the data access pattern is modified. A final optimization leads to an implementation that only needs one scalar variable to hold all stage vectors components of one time step one after another. This value is needed to update several data structures.

2.3. *Pseudocodes of multithreaded implementations*

In this subsection, pseudocodes for the main intermediate implementation versions of the RK method are given. Since the loop order of the imperfectly nested loops are main characteristics of the versions, the order of the loop indices are given as names to indicate the current loop nest.

2.3.1. *Vector version*

The straightforward implementation of the embedded RK scheme (2), (3) has the loops over the vector dimension as innermost loop. This has the advantage to provide a good spatial locality of the memory references, since the innermost loop implements the computations over vectors of length n. The initial implementation version can therefore be written in vector notation. Vectors are implemented as one-dimensional arrays and are written in boldface notation in the following. Omitting the error control and stepsize selection, the computations of one time step can be abbreviated as follows:

Version *i-l-j (Version* **1***):*

```
(1)  for ( $i = 0$; $i < s$; $i++$ ) {
(2)      $\mathbf{z} = \mathbf{0.0}$;
(3)      for ( $l = 0$; $l < i$; $l++$ )
(4)          $\mathbf{z} = \mathbf{z} + a[i][l] * \mathbf{v}_l$;
(5)      $\mathbf{z} = h * \mathbf{z} + \eta_\kappa$;
         barrier synchronization;
(6)      $\mathbf{v}_i = \mathbf{f}( x + c[i] * h , \mathbf{z} )$;
         barrier synchronization;
(7)  }
(8)  $\mathbf{z1} = \mathbf{0.0}$; $\mathbf{z2} = \mathbf{0.0}$;
(9)  for ( $i = 0$; $i < s$; $i++$ ) {
(10)     $\mathbf{z1} = \mathbf{z1} + bbs[i] * \mathbf{v}_i$;
(11)     $\mathbf{z2} = \mathbf{z2} + b[i] * \mathbf{v}_i$;
(12) }
(13) $\eta_{\kappa+1} = \eta_\kappa + h * \mathbf{z2}$;
(14) $\mathbf{err} = h * \mathbf{z1}$;
     barrier synchronization;
(15) error control;
```

The vectors **z**, **z1**, **z2**, **err** denote temporary vectors that are used to compute the arguments for the function evaluations and to compute the next approximation vector or the error vector. The computation order in the $i, l = 1, ...s$ iteration space is shown in Fig. 1 (top, left). The multithreaded implementation requires barrier synchronizations before and after the function evaluations as well as before the error control. The synchronization before the function evaluation is needed to ensure that the complete argument vector has been computed before any component of **f** is evaluated. The synchronization after the function evaluation is needed to avoid threads to continue updating vector **z** too early, since other threads might still need the previous values of all components of **z** for evaluating their components of **f**. The synchronization before the error control is required to ensure that the complete error vector **err** is available and all components of **err** can be used to decide whether the current time step should be accepted or needs to be repeated using a smaller step size. Moreover, **err** is used to compute the step size for the next time step. In summary, $2 \cdot s + 1$ barrier synchronizations are needed in each time step.

2.3.2. Separation of argument vectors

The first set of transformations modifies the computation of the argument vectors in program lines (2)–(5) of Version i-l-j: Separate argument vectors $\mathbf{z}[i]$ are introduced and replace $\mathbf{z}$ in the stage vector computations in order to decouple the dependencies. This modification will be exploited further for later loop restructuring. The initialization and the computation of the vectors $\mathbf{z}[i]$ are modified slightly so that the computation of the components of the argument vectors are implemented within a single nested loop. These transformations result in the program Version **2** with lines (1)–(6) replacing lines (1)–(7) of Version **1**.

Version *i-l-j-ext (Version 2):*

```
(1)   for ( i=0; i<s; i++ ) {
(2)       z[i]= $\eta_\kappa$;
(3)       for ( l=0; l<i; l++ )
(4)           z[i] = z[i] + h* a[i][l] * $\mathbf{v}_l$;
          barrier synchronization;
(5)       $\mathbf{v}_i$ = f( x + c[i] * h , z[i] );
(6)   }
      barrier synchronization;
(7)   z1 = 0.0; z2 = 0.0;
(8)   for ( i=0; i<s; i++ ) {
(9)       z1 = z1 + bbs[i] * $\mathbf{v}_i$;
(10)      z2 = z2 + b[i] * $\mathbf{v}_i$;
(11)  }
(12)  $\eta_{\kappa+1}$ = $\eta_\kappa$ + h* z2;
(13)  err = h* z1;
      barrier synchronization;
(14)  error control;
```

The computation order in the iteration space over the indices $i, l = 1, \ldots, s$ of the Butcher tableau is illustrated in Fig. 1 (top, right). The multithreaded implementation still needs a barrier synchronization before the function evaluation. However, no synchronization is needed after the function evaluation, since separate argument vectors are used so that no race conditions can occur. Another barrier is needed after the entire i-loop and before the error control. This results in $s + 2$ barrier synchronizations for each time step.

2.3.3. Loop interchange with dimension loop

The next version is a slight modification of Version **2**, which results by changing the loop structure in lines (3) and (4) of the code. More precisely, the implicit loop in code line (4) is made explicit and is interchanged with the l-loop in line (3). This results in the following implementation with the new lines (3) - (5):

Version *i-j-l-ext (Version 3):*

```
(1)   for ( i=0; i<s; i++ ) {
(2)       z[i]= $\eta_\kappa$;
(3)       for ( j=0; j<n; j++ )
(4)           for ( l=0; l<i; l++ )
(5)               $\mathbf{z}_j$[i] = $\mathbf{z}_j$[i] + h* a[i][l] * $(\mathbf{v}_l)_j$;
          barrier synchronization;
(6)       $\mathbf{v}_i$ = f( x + c[i] * h , z[i] );
(7)   }
      barrier synchronization;
```

```
(8)  **z1** = **0.0**; **z2** = **0.0**;
(9)  for ( i=0; i<s; i++ ) {
(10)     **z1** = **z1** + bbs[i] * $\mathbf{v}_i$;
(11)     **z2** = **z2** + b[i] * $\mathbf{v}_i$;
(12) }
(13) $\eta_{\kappa+1} = \eta_\kappa$ + h* **z2**;
(14) **err** = h* **z1**;
     barrier synchronization;
(15) error control;
```

The position of the barrier synchronizations remains unchanged.

2.3.4. Loop interchange of i and l loops

The next set of transformations are also based on program version **2** and applx two different transformations. Our aim to use stage vector components as soon after their computation as possible can be reached by interchanging the i-loop and the l-loop in the lines (1)–(6) of Version **2**. Starting with Version **2** several transformation steps have to be applied to lines (1)-(6) to get the equivalent implementation version with interchanged i-loop and l-loop. The resulting program version is the following:

Version *l-i-j:*

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

The computation order in the i, l plane of this intermediate code version is depicted in Fig. 1 (bottom, left). This version is not investigated independently, but it is used as basis for the next transformation.

A further improvement of the temporal locality of memory references is possible by a loop interchange with the innermost dimension loop. The innermost loop for the vector computations (now shown explicitly) and the loop for updating the argument vectors are interchanged and the resulting dimension loops are combined with the vector loops computing the vectors **z1** and **z2** by loop fusion. The resulting program exhibits no interleaved use of different stage vector components. Consequently, the stage vector computations can be represented by a single scalar variable *fx* that is used to store the components of all stage vectors one after another. This results in the following program version **l-j-i**:

Version *l-j-i (Version* **4***):*

```
(1)  for ( i=0; i<s; i++ )
(2)      z[i]= $\eta_\kappa$;
(3)  z1 = 0.0; z2 = 0.0;
     barrier synchronization;
(4)  for ( l=0; l<s; l++ ) {
(5)      for (j=0; j<n ; j++) {
(6)          fx= $f_j$ ( x + c[l] * h , z[l] );
(7)          $z1_j$ = $z1_j$ + bbs[l] * fx;
(8)          $z2_j$ = $z2_j$ + b[l] * fx;
(9)          for ( i=l+1; i<s; i++)
(10)             $\mathbf{z}_j$[i] = $\mathbf{z}_j$[i] + h* a[i][l] * fx;
(11)     }
         barrier synchronization;
(12) }
(13) $\eta_{\kappa+1}$ = $\eta_\kappa$ + h* z2;
(14) err = h* z1;
     barrier synchronization;
(15) error control;
```

The computation order in the iteration space is illustrated in Fig. 1 (bottom, right), omitting the dimension over the vector components. A barrier synchronization is needed before the l loop to ensure that all vectors **z**[i] are initialized before they are used for the following function evaluations. Similarly, a synchronization is needed after each iteration of the l loop to ensure that all components of vector **z**[l] are available for the evaluation of the components of **f** in the next iteration of the l loop. In summary, the same number of synchronizations in needed in each time step as for Versions **2** and **3**.

2.4. Parallel implementation and synchronization

For the four program versions described above have been implemented as multithreaded programs library the Pthread library. In the implementations, the computations of the components of the argument vectors, stage vectors, and approximation vectors are distributed in a block-wise way over different threads. The error control and stepsize selection for the next time step at the end of the previous time step is performed by a single thread. If the error control observes that the error is too large, the previous time step is repeated with a smaller step size [31]. Synchronization operations are included to ensure numerical correctness. A first barrier synchronization is used after the computation of each stage vector so that the computation of the next stage vector uses the most recent values of the preceding stage vectors. A barrier synchronization is also used before and after the error control and stepsize selection, which ensures that the approximation vectors $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ are completely computed before the error control and stepsize selection are performed and that all threads start the next time step not before the previous time step has been completed by all threads. Thus, the numerical behavior of the parallel versions and the sequential versions are identical.

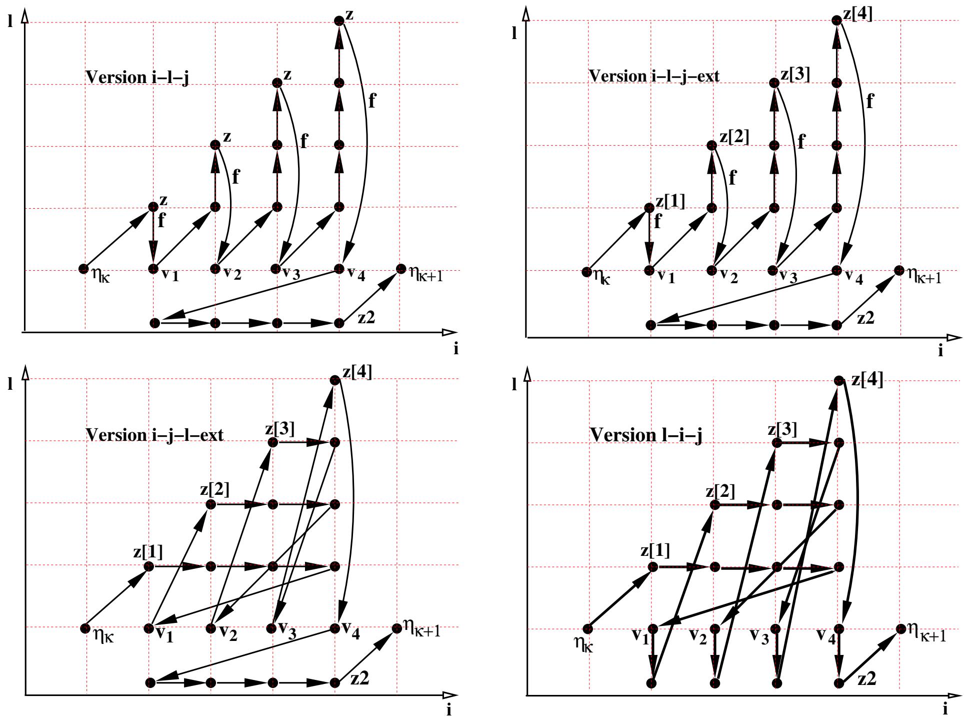


Figure 1: Illustration of the computation order in the iteration space of the i, l plane for the different program versions.

Comparing the number of synchronization for the different versions, each time step of Version 1 requires $2s+1$ barrier synchronizations where s is the number of stages of the RK method. In contrast, Versions 2 - 4 require only $s+2$ barrier synchronizations. Concerning the number of vectors required, the situation is the following: Version 1 has one vector **z** for computing the argument vectors and s stage vectors. Version 2 has s vectors $\mathbf{z}[i]$ for computing the argument vectors and s stage vectors. Version 3 also has s vectors $\mathbf{z}[i]$ for computing the argument vectors and s stage vectors, but uses a different access order. Version 4 has s vectors $\mathbf{z}[i]$ for computing the argument vectors, but only one vector for computing the stage vectors which reduces to one scalar value fx in the final Version 4 due to an optimized data access order.

3. Power and energy model

A power and energy model can help to identify the crucial factors of the power and energy consumption. After giving the terminology in Subsect. 3.1, the modeling of the power and energy consumption using DVFS is briefly described in Subsect. 3.2.

3.1. DVFS and frequency scaling factors

The energy $E = \int_{t=0}^{t_{end}} P(t)dt$ consumed for the execution of an application code depends on the execution time $T = t_{end}$ on the given hardware platform and the power drawing P during the execution. The power consumption P may vary during the execution. Analytical modes for the power consumption distinguish the dynamic power consumption P_{dyn} and the static power consumption P_{stat}. The dynamic power consumption is related to the supply voltage and the switching activity during the computing activity of the processor. The static power consumption is intended to capture the leakage power consumption as well as the power consumption of peripheral devices. The total power consumption is the sum of the dynamic power consumption and the static power consumption.

We consider DVFS processors with p_{max} cores and operational frequencies f between a minimum frequency f_{min} and a maximum frequency f_{max}. Frequency scaling for DVFS processors can be expressed by a dimensionless scaling factor $s \geq 1$, which describes a smaller frequency $f \leq f_{max}$ relative to the maximum frequency f_{max} as $f = f_{max}/s$. The maximum scaling factor is $s_{max} = f_{max}/f_{min}$. The power consumption P depends on the number of threads p used for the execution and the operational frequency f chosen.

3.2. Power and energy models for DVFS

The power model used in the following approximates the dynamic power consumption by $P_{dyn} = \alpha \cdot C_L \cdot V^2 \cdot f$, where α is the switching probability, C_L is the load capacitance, V is the supply voltage, and f is the operational frequency. The static power consumption is especially intended to capture the leakage power consumption which consists of several components, including sub-threshold leakage, reverse-biased-junction leakage, gate-induced-drain leakage, gate-oxide leakage, gate-current leakage, and punch-through leakage [24]. The exact power values for these components may vary and depend on the specific architecture considered. However, only approximations are needed for the power model. Such an approximation has been proposed by [7], modeling the static power consumption due to leakage power as $P_{stat} = V \cdot N \cdot k_{\text{design}} \cdot I_{\text{leak}}$, where V is the supply voltage, N is the number of transistors, k_{design} is a design dependent parameter, and I_{leak} is a technology-dependent parameter.

For DVFS processors, the power consumption depends on the operational frequency, which can be scaled within a predefined interval $[f_{min}, f_{max}]$. The following functional dependencies have to be considered: The frequency f depends linearly on the supply voltage V, i.e., $V = \beta \cdot f$ with some appropriate constant β. Thus, the dependence of the dynamic power consumption on the frequency f can be expressed as

$$P_{dyn}(f) = \gamma \cdot f^3 \tag{4}$$

with $\gamma = \alpha \cdot C_L \cdot \beta^2$. Usually, analytic power models are expressed as a function of the scaling factor $s = f_{max}/f$, which results in $P_{dyn}(s) = s^{-3} \cdot P_{dyn}(1)$ where $P_{dyn}(1)$ is the dynamic power consumption of the un-scaled case $s = 1$. This means that the dynamic power increases cubically when the operational frequency is increased, which can be used to study the change of the dynamic power consumption with respect to varying frequency values. Using $V = \beta \cdot f$ also for the static power consumption P_{stat} leads to a linear dependence of the static power on f, i.e., $P_{stat}(f) = \delta \cdot f$ with $\delta = N \cdot k_{\text{design}} \cdot I_{\text{leak}} \cdot \beta$ or $P_{stat}(s) = s^{-1} \cdot P_{stat}(1)$ where $P_{stat}(1)$ is the static power consumption in the un-scaled case. In total, the following equation results for the overall power consumption:

$$P_{total}(s) = s^{-3} \cdot P_{dyn}(1) + s^{-1} \cdot P_{stat}(1) \tag{5}$$

The reduction of the operational frequency of a processor by a scaling factor of s usually decreases the power consumption. However, it also increases the un-scaled execution time $T(1)$, leading to the scaled execution time $T(s)$. Assuming a constant part T_{const} that does not depend on the frequency scaling, the scaled execution time can be modeled as $T(s) = s \cdot T(1) + T_{const}$. The constant part T_{const} captures, for example, accesses to the main memory or to peripheral devices that are not affected by the scaling. Using the scaled execution time $T(s)$ and the modeled power consumption from Equation (5) yields the following model for the scaled energy consumption:

$$\begin{aligned} E(s) &= (P_{dyn}(s) + P_{stat}(s)) \cdot T(s) \qquad (6)\\ &= (s^{-3} \cdot P_{dyn}(1) + s^{-1} \cdot P_{stat}(1)) \cdot T(s) \end{aligned}$$

Equation (6) can be considered as a general model for the energy consumption of application codes depending on frequency scaling. This general model is the basis for the application-specific energy model to be developed in Section 5.

4. Experimental evaluation

The performance and energy behavior of the program versions derived in Section 2 are investigated in the following experimental evaluation using five multicore processors with different architecture. The evaluation considers the execution time, the power consumption P, and the energy consumption E. Since the energy E is defined as $E = \int_{t=0}^{t_{end}} P(t)dt$, assuming that the program is executed from time $t = 0$ to time $t = t_{end}$, the power consumption $P(t)$ at time t may have a strong influence on the overall energy consumption.

4.1. Experimental setup

Five multicore systems with different architectures (Haswell, Broadwell, Skylake, Coffee Lake, Cascade Lake) have been used for the experimental evaluation, see Table 1 for an overview of the characteristics of the systems. Three of the processors (Haswell, Skylake, Cascade Lake) and two are desktop processors (Broadwell, Coffee Lake). The compilation of the different program versions has been performed with gcc (version 4.8.5 on Broadwell, version 4.3.4 on Xeon, and version 7.5.0 on the other systems). The optimization level `-O3` has been used, which ensures that the compiler can perform all useful transformations for the different RK versions.

Two ODE systems with different computational demands have been chosen as application problems to be solved by the RK methods: (i) The Brusselator ODE system [20], results from a spatial discretization of a two-dimensional time-dependent partial differential equation describing a reaction-diffusion problem of two chemical substances. Different discretization lead to different sizes of the resulting ODE system: Using N discretization points in each space dimension leads to an ODE system of size

	Xeon E5 2690	**Xeon i7 6950X**	**Core i9 9980XE**	**Core i7 9700**	**Xeon Gold 6248**
architecture	Haswell	Broadwell	Skylake	Coffee Lake	Cascade Lake
year of release	2014	2016	2018	2019	2019
minimum frequency	1.8 GHz	1.2 GHz	1.2 GHz	0.8 GHz	1.0 GHz
maximum frequency	2.9 GHz	3.0 GHz	3.0 GHz	3.0 GHz	2.5 GHz
TDP	130 W	140 W.	165 W	65 W	150 W
physical cores	24	10	18	8	20
hyperthreading	no	yes	yes	no	no
L1 data cache	32 KB	32 KB	32 KB	32 KB	32 KB
L2 cache	256 KB	256 KB	1 MB	256 KB	1 MB
L3 shared cache	30 MB	25 MB	25 MB	12 MB	28 MB
RAM size	64 GB	32 GB	64 GB	16 GB	376 GB

Table 1: Characteristics of the processors used for the experimental evaluation.

$n = 2N^2$. The resulting Brusselator ODE system has the property that each component of the right-hand side function **f** has a constant evaluation time that is independent of the size of the ODE system. Thus, the evaluation time of the entire function **f** increases linearly with n. (ii) The Schrödinger–Poisson ODE system results from applying a spectral method to a time-dependent 1D partial differential equation describing the behavior of a collisionless electron plasma. The resulting Schrödinger–Poisson ODE system has the property that the evaluation time of each component of **f** increases linearly with n. Thus, the evaluation time of the entire function **f** increases quadratically with n. The different program versions are used to solve these ODE systems. As example for an ODE solution method, the popular DOPRI5 method has been used for all experiments. We expect that other RK methods lead to similar results.

The time and energy measurements have been performed using the Running Average Power Limit (RAPL) interface and sensors of the Intel architecture [37, 22]. RAPL sensors can be accessed by control registers, known as Model Specific Registers (MSRs), which are updated in intervals of about 1 ms [22]. To access the MSRs, we have used the likwid tool-set, especially the likwid-powermeter and the likwid-perfctr tools in Version 4.1 (Haswell and Broadwell) or Version 5.1 (remaining systems)[43]. For all measurements, the threads are pinned to dedicated cores using likwid-pin. Experiments have shown that the energy measurement with RAPL sensors are quite accurate when compared to measurements with power-meters [37, 36].

4.2. *Experiments with different ODE problems*

Different ODE problems may have different computational requirements due to the evaluation cost of the right hand side function **f**. This may strongly influence the execution time and energy behavior of the parallel implementation versions.

4.2.1. *Experiments with the Schrödinger ODE*

Figures 2, 3 and 4 shows the execution time, energy consumption, and power consumption resulting by executing 50 time steps of the DOPRI5 RK method for a

Schrödinger ODE system with discretization $n = 5000$ on two different hardware systems. The number of threads is increased until the total number of cores of the system used has been reached (24 on the Haswell system and 20 on the Cascade Lake system, see Table 1). The values for $p = 1$ are the values for a pure sequential implementation without any thread overhead. For all measurements, the complete application has been considered. This includes the allocation and initialization of the data structures, which is performed by the master thread alone for the parallel versions before the generation of the remaining threads, and, for the parallel versions, the generation of the threads. Figure 2 shows that the execution time decreases with an increasing number of threads for all versions on both systems, i.e., the method shows a good scalability on both systems. On the Haswell architecture using 24 threads, the speedup reached is 15.4. On the Cascade Lake architecture, the speedup obtained with 20 threads is 14.0. All versions show a very similar behavior, i.e., the differences in the different implementation versions are covered by the large evaluation cost of the right hand side Schrödinger function. The good scalability can be explained by the fact that the Schrödinger ODE system has a right hand side function with high evaluation cost, which can be perfectly distributed among the threads. Comparing the two hardware systems, it can be seen that execution times on the Haswell system are smaller than the execution times on the Cascade Lake system due to the faster clock rate of the Haswell system.

Figure 3 shows that the energy consumption decreases with an increasing number of threads on both hardware systems. This decrease can be explained by the fact that idle cores also consume power and energy during their idle time, however they do not contribute to the computation of the result. Correspondingly, the evaluation time for the application is longer if fewer cores are used, and thus the energy consumption increases. Comparing the Haswell and Cascade Lake system, it can be seen that Cascade Lake system requires less energy than the Haswell system for the same number of threads. This can be attributed to the smaller power consumption of the Cascade Lake system, see Fig. 4, which is caused by the smaller clock frequency used and the cubic dependence of the dynamic power consumption on the clock frequency (see Equ. 4). Additionally, the Cascade Lake processor has energy saving features that have not yet been used for the Haswell processor, see [39] for a detailed treatment. The increase of the power consumption with the number of threads can be explained by the fact that idle cores use less energy than cores that contribute to the computation. Therefore, the power consumption increases if the number of participating cores increases. This can be observed in Fig. 4 for all versions.

Table 2 summarizes the performance and energy characteristics of the different hardware systems when solving the Schrödinger ODE with discretization $n = 5000$. For all hardware systems, the maximum energy consumption results for a sequential execution, employing a single hardware core. Similarly, the minimum energy consumption and the minimum execution time results when employing all hardware cores. For the hardware systems with hyperthreading, no significant reduction of execution time and energy consumption can be obtained when starting more threads than there are hardware cores. Thus, the usage of hyperthreading does not lead to an advantage, which can be expected due to the compute-intensive behavior of the Schrödinger ODE. For all hardware systems, a sequential execution leads to the minimum power consumption, and the maximum power consumption results when employing all hard-

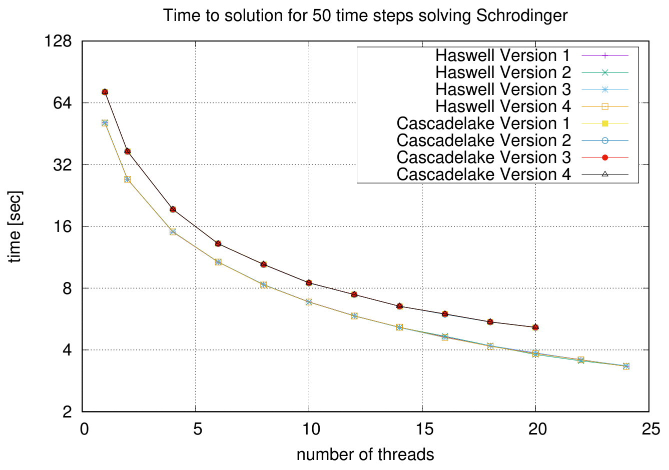


Figure 2: Parallel execution time of the different RK versions applied to 50 steps of the Schrödinger ODE for system size n=5000 on the Haswell and the Cascade Lake architecture.

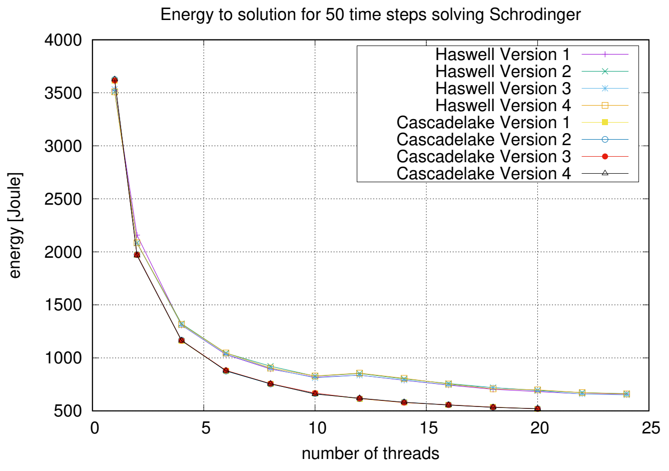


Figure 3: Parallel energy consumption of the different RK versions applied to 50 steps of the Schrödinger ODE for system size n=5000 on the Haswell and the Cascade Lake architecture.

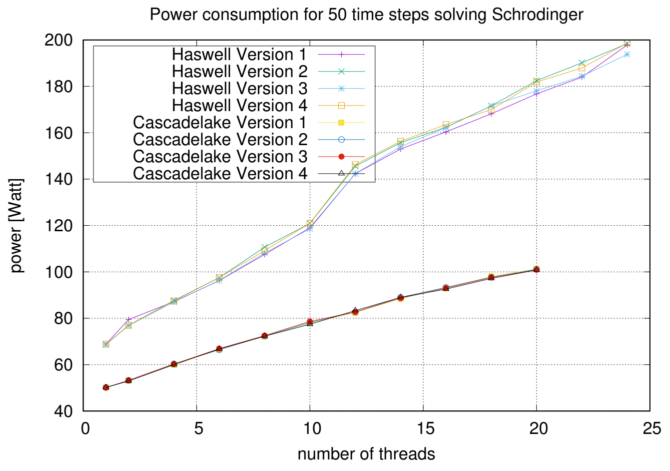


Figure 4: Parallel power consumption of the different RK versions applied to 50 steps of the Schrödinger ODE for system size n=5000 on the Haswell and the Cascade Lake architecture.

	Haswell	**Broadwell**	**Skylake**	**Coffee Lake**	**Cascade Lake**
number of threads	24	10	18	8	20
speedup	**15.4**	8.3	14.2	7.2	14.2
minimum time [s]	**3.32**	7.6	4.2	8.3	5.1
maximum time [s]	**51.3**	63.4	59.9	60.0	72.3
minimum energy [J]	649.9	468.3	472.3	**331.8**	519.3
maximum energy [J]	3533.6	605.3	2394.5	**564.2**	3626.1
minimum power [W]	68.7	9.5	37.9	**8.9**	50.1
maximum power [W]	198.8	62.8	112.8	**39.9**	101.3

Table 2: Performance, energy and power characteristics of the Schrödinger ODE on different processors.

ware cores. Comparing the different hardware systems, it can be observed that the server processors (Haswell, Skylake, Cascade Lake) have a significantly larger power consumption as well as a significantly larger maximum energy consumption than the desktop processors (Broadwell, Coffee Lake).

4.2.2. Experiments with the Brusselator ODE

The situation changes if the Brusselator ODE system is considered, see Figures 5, 6, and 7 for the resulting execution time, energy consumption and power consumption on three different hardware systems (Haswell, Skylake, Coffee Lake) when executing 50 time steps for the Brusselator ODE system using discretization N=4096. For this ODE system, differences between the different implementation versions can be observed, since the evaluation time of the right hand side function of the Brusselator ODE system is quite small and the differences in the memory accesses of the different implementation versions become more relevant.

Figure 5 shows that the execution time still decreases for an increasing number of threads. However, the decrease is much smaller than for the Schrödinger ODE system

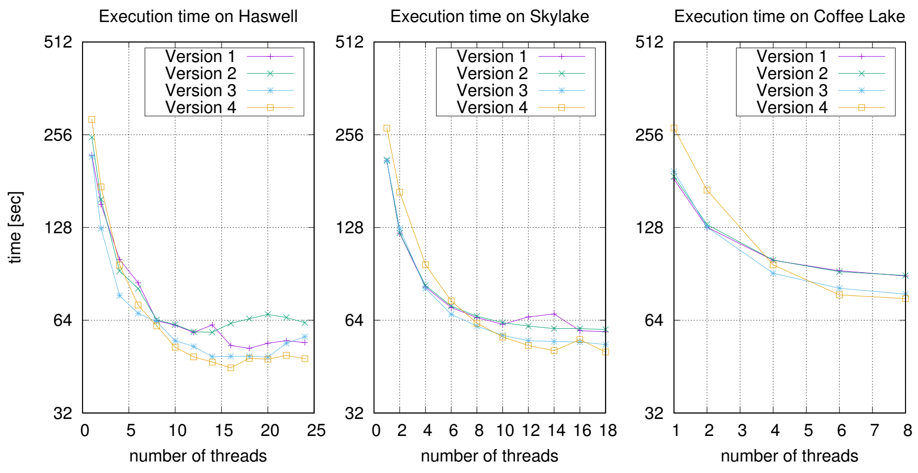


Figure 5: Execution times of the different RK versions applied to 50 steps of the Brusselator ODE using different numbers of threads for system size N=4096 on Haswell (left), Skylake (middle), and Coffee Lake (right).

and different hardware systems show slightly different effects: On the Haswell system, the smallest execution times are obtained if not all hardware cores are employed. On the Skylake and the Coffee Lake systems, the smallest execution time results when all hardware cores are used. Using $p = 24$ threads on the Haswell system, the largest speedup observed is for Version **4**, which reaches a speedup of 6.0. For comparison, Version **1** reaches a speedup of only 4.1.

Comparing the sequential versions, it can be seen in Figure 5 that Versions **1** and **3** are the fastest sequential versions on all three hardware systems. Version 4 has a sequential execution time which is significantly larger than the sequential execution time of Version **3**. The situation changes when the number of threads is increased, and when using all hardware cores, Version 4 is the fastest version on all three hardware systems. For example, on the Haswell system Versions 1 and 3 have execution times which are 12.8 % and 17.8 %, respectively, larger than the execution time of Version **4**.

Comparing Versions **1** and **2**, it can be seen that these versions have similar execution times on the Skylake and the Coffee Lake systems, as well as on the Haswell system for up to 14 threads. When using more than 14 threads of the Haswell system, Version **2** is slower than Version **1**. Thus, Version **2** cannot take advantage of the reduced number of barrier synchronizations. Instead, the increase of the data volume of Version **2** caused by the additional vectors outweighs the reduction caused by the smaller number of synchronizations. On all three hardware systems it can be observed that Versions **3** and **4** are the fastest versions when more than half the number of the available hardware cores are employed. This effect is caused by the better scalability of these two versions.

The energy consumption shows a slightly different behavior than the execution time, see Figure 6. For the Haswell and Skylake systems, the energy consumption de-

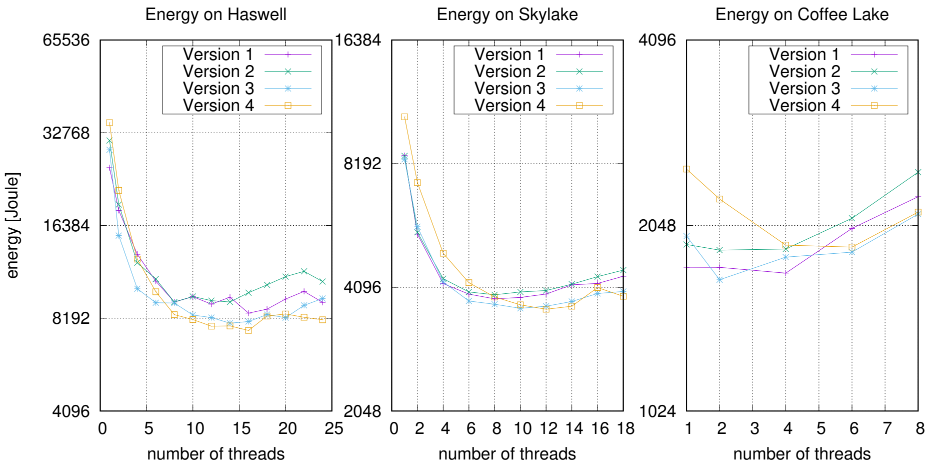


Figure 6: Energy consumption of the different RK versions applied to 50 steps of the Brusselator ODE using different numbers of threads for system size N=4096 on Haswell (left), Skylake (middle), and Coffee Lake (right).

creases with the number of threads until a certain number of threads is reached (about 14 threads on Haswell and about 10 threads on Skylake). When using more threads, the energy consumption stays stable or may even increase, depending on the implementation version and the hardware system. For the Coffee Lake system, no reduction of the energy consumption with the number of threads can be observed. Comparing the amount of energy consumed, it can be seen that the two server systems (Haswell and Skylake) consume significantly more energy than the desktop system (Coffee Lake). Comparing the different implementation versions, it can be observed that for a sequential execution, Version 1 has the smallest energy consumption, especially on the Haswell and the Coffee Lake systems. On these two systems, the sequential energy consumption of Version 4 is about 40 % larger than the sequential energy consumption of Version 1. This difference decreases with the number of threads. On all three systems, either Version 3 or 4 lead to the smallest energy consumption when using a larger number of threads. The difference in the behavior of the execution times and the energy consumption can be explained by the increase of the power consumption with the number of threads, see Figure 7. In particular for a larger number of threads, the reduction of the execution time when using more threads is over-compensated by the increase of the power consumption, leading to a stagnation or a slight increase of the energy consumption.

Similar observations can be made for the other two processors that are not covered in the diagrams. For the Broadwell system, the advantage of Version **4** over Versions **1** and **2** is even larger. This can also be seen in the next section. Table 3 gives information about performance and energy consumption for all hardware systems considering different numbers of threads up to the number of hardware cores available. Comparing Tables 2 and 3 shows that the Brusselator ODE usually leads to smaller speedup val-

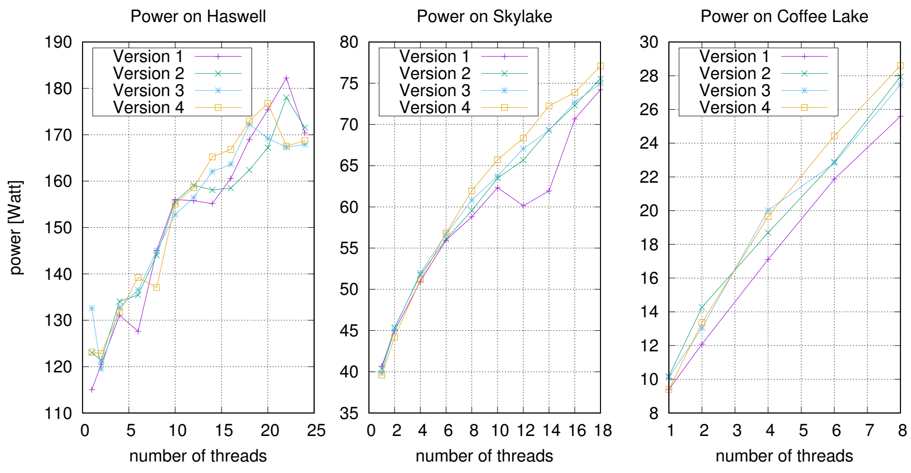


Figure 7: Power consumption of the different RK versions applied to 50 steps of the Brusselator ODE using different numbers of threads for system size N=4096 on Haswell (left), Skylake (middle), and Coffee Lake (right).

	Haswell	**Broadwell**	**Skylake**	**Coffee Lake**	**Cascade Lake**
number of cores	24	10	18	8	20
maximum speedup	6.0	**10.4**	5.3	3.6	7.2
minimum time [s]	48.0	45.7	50.5	75.3	**42.6**
maximum time [s]	287.0	620.8	**268.0**	269.1	315.8
minimum energy [J]	8100.4	2154.9	3622.6	**1713.9**	3753.43
maximum energy [J]	35333.5	5534.4	10651.8	**2528.4**	16535.8
minimum power [W]	115.0	**8.9**	39.6	9.4	52.3
maximum power [W]	176.8	59.2	77.1	**28.6**	88.6

Table 3: Performance, energy and power characteristics of the Brusselator ODE on different processors.

ues than the Schrödinger ODE, since the Brusselator ODE is less compute-intensive than the Schrödinger ODE. The large speedup for the Broadwell systems is caused by caching effect of Version 4 that lead to large sequential execution times. On all systems, the smallest execution times are caused either by Versions 3 or Version 4. In contrast to the Schrödinger ODE, hyperthreading can help to further reduce the execution times of the Brusselator ODE (not covered in the table). Again, it can be seen that the server processors (Haswell, Skylake, Cascade Lake) have a much larger power and energy consumption than the desktop processors (Broadwell, Coffee Lake). This effect is especially observable for the newer Coffee Lake system.

4.3. Experiments with frequency scaling

For the experiments with frequency scaling, the two processors that provide hyperthreading have been selected (Broadwell, Skylake). The number of threads used for the execution has been set such that each thread can be mapped to a different logical core, i.e., 20 threads are used for the Broadwell processor and 36 threads for the Skylake processor. The operational frequencies of the cores are set to a fixed value using `likwid-setFrequencies`. The uncore frequencies are not set explicitly. Figures 8, 9 and 10 depict the execution time, the energy consumption and the power consumption of the different program versions from Section 2 on the Broadwell processor (left) and the Skylake processor (right) for all possible frequencies that the processors provide.

For the Broadwell processor, it can be observed that Version 3 has a smaller execution time and a smaller energy consumption than Version 1 and 2 for all frequencies, although the difference is quite small. The behavior of Version 4 is most interesting: For small frequencies, it has the largest execution time. However, when increasing the frequency, the execution time decreases faster than for the other versions, and for frequencies larger than 2.1 GHz, Version 4 is the fastest version. Moreover, for operational frequencies larger than 1.5 GHz, Version 4 uses the smallest amount of energy, although the difference to Version 3 is not very large, see Fig. 6 (left). For all versions, the smallest energy consumption results when using the smallest frequency available, which is 1.2 GHz. The power consumption increases steadily when increasing the operational frequency, see Figure 10 (left). For 3.0 GHz, Version 4 uses only about 79.5 % of the execution time of Versions 1 and 2. For the same frequency, the energy consumption is only reduced to 85.5 %. The difference in savings of execution time and energy is caused by the fact that Version 4 has a larger power consumption than Versions 1 and 2 for frequencies larger than 2.5 GHz.

For the Skylake processor, the situation is slightly different. For all frequencies, Versions 3 and 4 have a smaller execution time than Versions 1 and 2, and Version 4 is slightly faster than Version 3, see Figure 8 (right). For 3.0 Ghz, Version 4 has about 87 % of the execution time of Version 1, i.e., the percentage advantage is smaller than for the Broadwell processor. Versions 3 and 4 consume less energy than Versions 1 and 2, see Figure 9 (right). For 3.0 Ghz, the percentage energy advantage of Version 4 over Version 1 is about 6 %. For smaller frequencies, the advantage is larger (up to 10 %). For all versions, the smallest energy consumption is not obtained for the smallest frequency. Instead, Versions 3 and 4 have their smallest energy consumption for 1.4 GHz. The energy consumption of all version shows only slightly variations for frequencies up to 2.2 GHz and rises significantly for the two largest frequencies 2.9 GHz and 3.0 GHz.

Comparing the Broadwell and the Skylake processor, it can be seen that the Skylake processor has a significantly larger energy consumption than the Broadwell processor, see Figure 9, although the execution times are quite similar, see Figure 8. This effect is caused by the significantly larger power consumption of the Skylake processor, see Figure 10.

4.4. Analysis of the power consumption

The diagrams in Figures 4, 7, and 10 have shown the average power consumption observed for the entire execution of the different RK versions. However, there are variations of the power consumption during program execution that are now analyzed in more detail by considering the dynamic development of the power during the course of the execution of the RK versions. In particular, Versions 1 and 4 are compared.

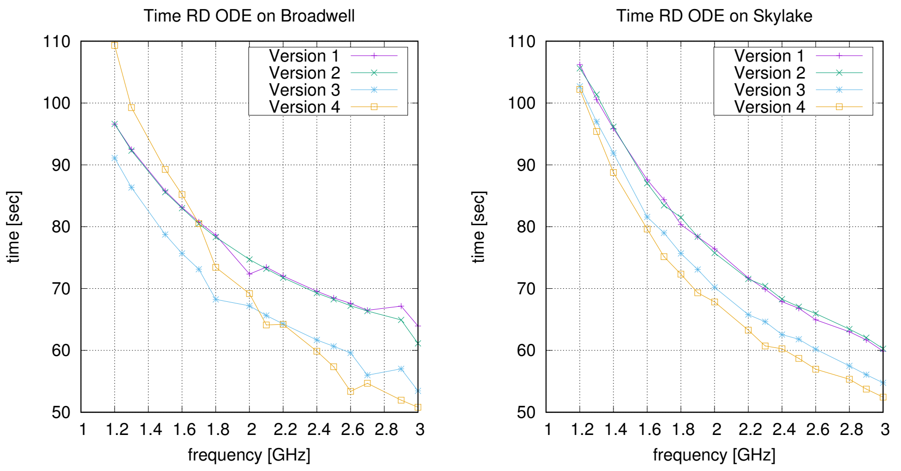


Figure 8: Parallel execution time of RK methods applied to 50 steps of the Brusselator ODE for system size N=4096 on Broadwell (left) using 20 threads and Skylake (right) using 36 threads.

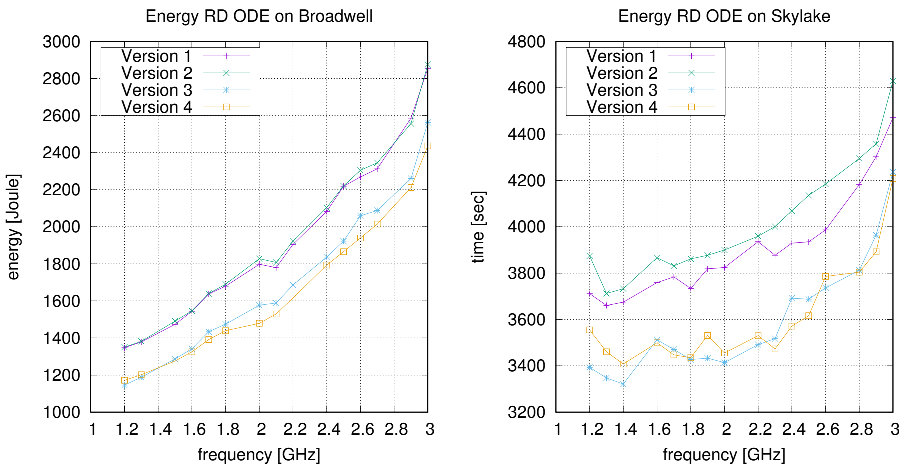
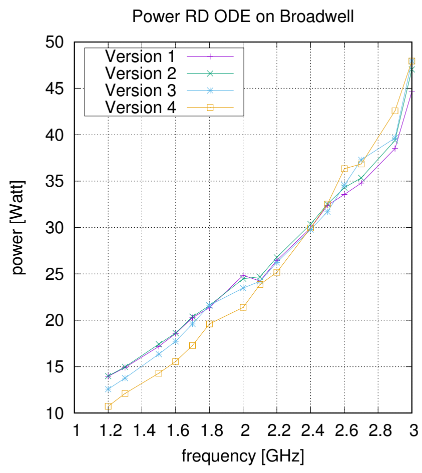


Figure 9: Parallel energy consumption of RK methods applied to 50 steps of the Brusselator ODE for system size N=4096 on Broadwell (left) using 20 threads and Skylake (right) using 36 threads.



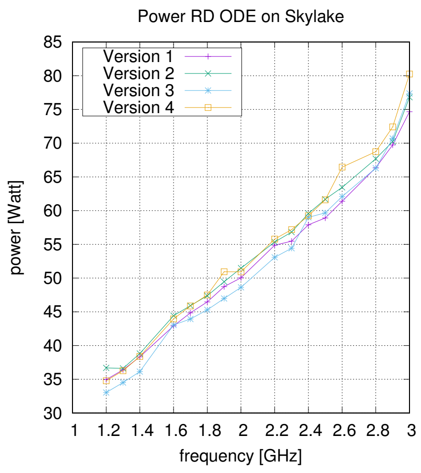
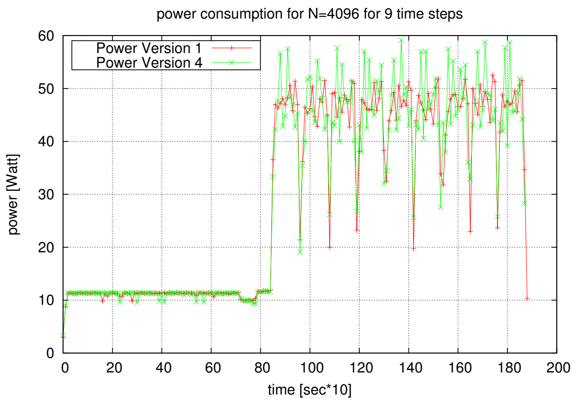


Figure 10: Parallel power consumption of RK methods applied to 50 steps of the Brusselator ODE for system size N=4096 on Broadwell (left) using 20 threads and Skylake (right) using 36 threads.

Figure 11 shows the dynamic development of the power consumption of Version 1 (red lines) and Version 4 (green lines) for the Brusselator ODE system with discretization N = 4096 on the Broadwell processor. The diagrams are obtained by monitoring the execution of the entire program using the timeline mode of `likwid-perfctr` with time frequency 100 ms. The execution has been performed with the highest operational frequency available. The upper and the lower diagram, respectively, show the power consumption for 10 threads (on 10 cores) and for 20 threads (on 10 cores using hyperthreading).

Each of the RK versions consists of an initialization phase that allocates the data structures used for the computations and the actual computation phase that executes the time steps of the RK method. The initialization phase is executed sequentially by a single thread that starts the working threads after the initialization. The specific integration interval used for the ODE system leads to the execution of nine consecutive time steps of the RK method in the computation phase. In Figure 11, the initialization phase and the computation phase of the program can clearly be distinguished. Due to the sequential execution, the initialization phase takes the same amount of time in both diagrams and for both RK versions depicted. The power consumption in the initialization phase is about 11.5 Watt.

The following computation phase is executed by all worker threads and performs nine time steps. These nine time steps can clearly be identified in the power diagrams by the peaks in the power development, especially in the lower diagram. The end of the different time steps can be identified by a drop in the power consumption, which is caused by the sequential execution of the error control and the stepsize selection. Version 4 has larger variations in the power consumption than Version 1 during the computation phase, which leads to an overall larger average power consumption for the entire execution. For 10 threads, it can be observed that both versions 1 and 4



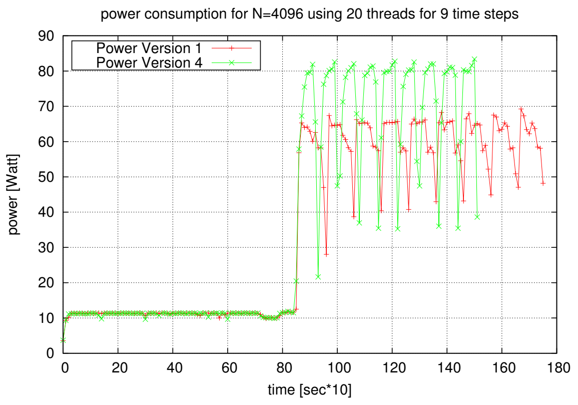


Figure 11: Development of the power consumption during program execution for 10 threads (top) and 20 threads (bottom) on the Broadwell processor when solving the Brusselator ODE.

need about the same time for the execution of the computation phase. For 20 threads, Version 4 executes the computation phase significantly faster than Version 1, which leads to a smaller overall execution time, see also Fig. 8. This demonstrates that the loop structure of Version 4 provides a larger potential of parallelism, which can be exploited by the additional threads.

The RK versions perform the same arithmetic operations (potentially in a different order), i.e., the difference in the performance and energy behavior observed can only be caused by differences in the memory access behavior. In particular, the faster execution time of Version 4 over Version 1 can be explained by a better utilization of the memory hierarchy due to the better temporal locality of the memory accesses caused by the loop transformations described in Section 2.2. This can be seen when considering the utilization of the L2 cache: Figure 12 compares the development of the load and evict volumes of the L2 cache during the execution of Versions 1 and 4 for 20 threads. Again, the Brusselator ODE is solved for the same situation as in Figure 11. The diagram shows that Version 4 performs significantly more accesses to the L2 cache than

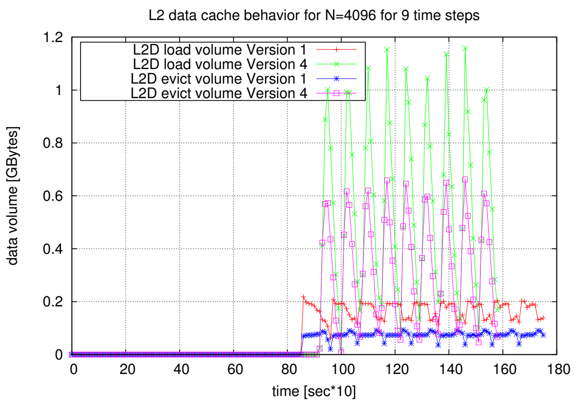


Figure 12: Development of the L2 load volume during program execution on the Broadwell processor for solving the Brusselator ODE for versions 1 and 4.

Version 1 during the execution of the computation phase, indicating a more efficient use of the L2 cache. This is also supported by the fact that the main memory the L3 data volumes of Version 4 is significantly smaller than that of Version 1. Detailed measurements with `likwid-perfctr` (not shown in a diagram) show that Version 4 leads to only 60.5%/87.3% of the main memory read/write volume of Version 1. Similar observation can be made for the L3 cache: Version 4 leads to only 69.5%/52.5% of the L3 read/write volume of Version 1.

4.5. Analysis of the memory access behavior

The preceding subsection has indicated that the memory access behavior may play an important role for the difference in the performance and energy behavior of the different RK versions. This is now considered by a more detailed analysis using the DVFS scaling factor s from Section 3. Using frequency scaling factor s, the overall execution time $T^V_{total}(n, s)$ of an RK version V for solving an ODE system of size n using p threads can be expressed by the following runtime function:

$$T^V_{total}(n, s, p) = T^V_{mem}(n, p) + T^V_{cache}(n, s, p) + T^V_{op}(n, s, p) + T^V_{sync}(s, p) \qquad (7)$$

with the following components:

- $T^V_{mem}(n, p)$ captures the total access time of Version V to the main memory. This time depends on the system size n, but it is independent from the scaling factor s, since the memory access time is not affected by DVFS. Only the core frequency has been varied in the experiments described in Section 4.3. Different implementation versions may have different memory access times due to differences in their memory access pattern.

- $T^V_{cache}(n, s, p)$ captures the total access time of version V to the caches of the different levels. This time depends on n, p, and s, since the cache access times are affected by DVFS, i.e., the cache access time increases with the scaling factor.

Different versions may lead to different cache access times due to differences in the memory accesses performed.

- $T_{op}^V(n, s, p)$ captures the time for performing arithmetic operations. This time is influenced by n, p, and s. Since all versions perform the same number of numerical operations (possibly in a different order), the same time for $T_{op}^V(n, s, p)$ for fixed n, p, and s results, independently from the specific version.

- $T_{sync}^V(s, p)$ captures the time needed for synchronizations that are required for a parallel execution. Different versions have different synchronization requirements, see Section 2.3, and the synchronization time depends on the number p of threads and the scaling factor s.

For large values of n and moderate values of p, $T_{sync}^V(s, p)$ is small compared to the other components and is not considered further. In the following, we consider T_{mem}^V and T_{cache}^V in more detail, since these terms are mainly responsible for the performance and energy differences between the different versions. The memory and cache access volumes (in GBytes) of the different versions are shown in Table 4, considering the parallel execution with 20 threads on the Broadwell processor for the Brusselator ODE with 50 time steps and discretization $N = 4096$. The information in the table has been obtained using likwid-perfctr to access the corresponding MSR registers. Experiments with a sequential execution or a parallel execution with a different number of threads lead to the same volumes of memory read and write accesses. The information in Table 4 shows the following differences of the RK versions:

- **Main memory:** Version 3 and especially Version 4 exhibit a significantly smaller read volume from the main memory (MM) than Versions 1 and 2. Versions 3 and 4 have been designed with the goal to have a better utilization of the memory hierarchy by a program restructuring that leads to a larger potential of temporal locality. The measurements show that this goal has been reached by Versions 3 and 4. The difference between the versions for the write volume from main memory is smaller.

- **L3 cache:** The usage of the L3 cache shows a similar behavior than the usage of the main memory: Versions 3 and 4 lead to less load and evict operations than Versions 1 and 2. However, the difference is not as large as for the main memory. The biggest difference can be observed for the L3 evict volume.

- **L2 cache:** The situation is different for the usage of the L2 cache: Table 4 shows that Version 4 has a significantly larger load and evict volume of the L2 cache than the other versions, which indicates a more intense use of the L2 cache due to an increased temporal locality. Compared to Versions 1 and 2, Version 3 also leads to less load and evict operations.

The differences observed in the memory access behavior of the different versions can help to explain differences in the execution times for the Broadwell processor as well as for the other processors, see Figure 5. For a sequential execution of Version 4, the reduction in the main memory and L3 cache data volumes of Version 4 cannot compensate the large L2 data volumes. However, when increasing the operational frequency, the difference in the execution time decreases, which can be explained by the fact that the time for the L2 data accesses decreases when increasing the frequency, but the time for the main memory operations remains unchanged. In the sequential case, this effect is not large enough to reduce the time for load operations of Version 4 below those of Versions 1 to 3. Therefore, Version 4 remains significantly slower than the other versions in the sequential case.

For a parallel execution, the access operations to the L2 cache are distributed between the different threads, which reduces the time for the L2 load operations. However, for small operational frequencies the time for the access operations to the L2 cache is quite high, so that the savings in memory read and write volume are not compensated. When increasing the frequency, the reduction in the access time to the L2 cache leads to a larger saving in time, and in combination with the smaller volume of memory operations, Version 4 outperforms Versions 1 and 2. This effect increases with the number of threads, which can be explained by the decreasing number of L2 operations per thread. The execution time of Version 3 is smaller than the execution time of all other versions for smaller frequencies up to 2.1 GHz, see Fig. 8. For frequencies over 2.1 GHz, Version 4 has a smaller execution time than Version 3. This behavior can be explained by the memory and cache access behavior, see Table 4. The amount of read accesses to the main memory and accesses to the L3 cache of Version 3 lies between the amount of Version 1 and 2 on the one hand and the amount of Version 4 on the other hand. Therefore, Version 3 is faster than all other versions for small frequencies, and for larger frequencies, Version 4 has the smallest execution time due to the L2D frequency scaling behavior described.

Table 4: Memory access behavior for the Brusselator ODE on the Broadwell processor

Version	Version 1	Version 2	Version 3	Version 4
MM read volume [GBytes]	1458.4	1459.2	1256.6	**882.8**
MM write volume [GBytes]	746.7	746.7	**545.3**	652.5
L3 load volume [GBytes]	1646.4	1657.3	1446.2	**1144.5**
L3 evict volume [GBytes]	395.1	390.9	293.9	**207.4**
L2D load volume [GBytes]	1720.9	1741.5	1545.1	**4269.3**
L2D evict volume [GBytes]	751.3	755.2	551.8	**2526.8**

4.6. Summary of performance, energy and power behavior

The experimental evaluation for the different processors has shown that application-specific loop transformations can have a significant impact on the energy consumption and the performance of the resulting program versions. The specific impact observed depends on different influencing factors, including the processor architecture, the number of threads used for the execution, the operational frequency, the specific ODE problem to be solved, and the size of the input data for the ODE problem. The following observation can be made:

Processor architecture: Comparing the five processors used, it can be seen that there are slight differences in the execution times for the same given ODE system and a specific system size, see the minimum and maximum execution time entries in Tables 2 and 3. When using the same number of threads for the Brusselator ODE, the Coffee Lake processor is about 30 % slower than the other processors. However, the execution times on the different processors are quite similar for the Schrödinger ODE. The differences of the energy consumption between the different processors can be quite larger: the two desktop processors (Broadwell, Coffee Lake) require much less energy than the three server processors (Haswell, Skylake, Cascade Lake). For example when using eight threads, the Cascade Lake processor uses between 50 % and 75 % more energy than the Coffee Lake processor, depending on the ODE problem and the implementation version. The reason for this behavior lies in the fact that the desktop processors have a significantly smaller power consumption than the server processors.

Number of threads: The execution time typically decreases and the power consumption typically increases when the number of threads is increased. This can be observed for all five hardware systems, for both ODE applications and all four program versions. For the Brusselator ODE, the scalability of Version 4 is significantly better than the scalability of the other versions. Although Version 4 starts with a slower sequential execution time in most cases, Version 4 outperforms the other versions when using a larger number of threads. This observation can be made for all processors considered, but there are differences in the quantitative behavior of the different processors, and the percentage improvement in the execution time of Version 4 over Version 1 varies between the processors.

Frequency scaling: Reducing the operational frequency reduces the dynamic power consumption of the cores. On the other hand, the execution time is increased, and the effect on the energy consumption results from a combination of these two effects. For all platforms and implementation versions, using the highest frequencies results in the largest energy consumption, which is usually significantly larger than the smallest energy consumption. However, using the smallest frequency provided does not always lead to the smallest energy consumption. Instead, there might be another (small) frequency that has a slightly smaller energy consumption. This effect could, e.g., be observed for the Skylake processor, see Figure 9.

Application problem: For ODE applications with an expensive right-hand side function (such as Schrödinger), there is no significant difference in the time and energy behavior of the four versions, and all version show a very good scalability. For ODE applications with a less expensive right-hand side function (such as Brusselator), the execution time varies with the system size and the number of threads. For small system sizes and a small number of threads, Versions 1 and 3 are often the most efficient ones. For large systems and a larger number of threads, Versions 3 and 4 are the best implementations, and Version 4 usually has a slight advantage over Version 3.

5. Modeling of power and energy consumption

The observed performance and energy behavior of the different RK versions can be modeled using the power and energy models from Section 3. This will be considered in this Section in more detail.

5.1. Power modeling

The power model in Equ. (5) uses two parameters $P_{dyn}(1)$ and $P_{stat}(1)$ expressing the dynamic and static power consumption for the un-scaled case $s = 1$. A parameter fitting approach can be used to determine the values of these two parameters using power values that have been measured for the different RK versions on a specific processor. The parameter fitting has been performed using the gnufit tool, which is based on a nonlinear least squares fit mechanism based on the Marquardt-Levenberg-algorithm. We consider two functions for the fitting: The first function is Equ. (5) and the data values are the power values measured for the four different RK versions for different frequencies. An alternative power function to Equ. (5) is

$$P^{alt}(s) = s^{-3} \cdot P^{alt}_{dyn}(1) + P^{alt}_{stat}(1). \tag{8}$$

This function is based on the assumption that the static power consumption is constant and does not depend on the operational frequency, which has been proposed by several authors [48]. Table 5 shows the result of the modeling for the two power functions (5) and (8) and the resulting parameter values. The table also shows the root-mean-square deviation (RMSD), which measures the standard deviation between the predicted and the measured power values, i.e., smaller RMSD values indicate a better fit than larger RMSD values. The RMSD values are computed as $\sqrt{WSSR/ndf}$, where $WSSR$ is the weighted sum of squared residuals, also referred to as chi-square, and ndf is the number of degrees of freedom, which is 13 in our case, since 15 different frequencies (or scaling factors) are investigated and two parameters are to be determined.

Table 5: Power modeling for the RD ODE on the Broadwell processor

Version	Version 1	Version 2	Version 3	Version 4
$P_{dyn}(1)$[W]	10.56	12.69	16.71	24.23
$P_{stat}(1)$[W]	31.34	30.65	27.77	22.36
RMSD (5)	1.09	1.35	1.37	0.76
$P^{alt}_{dyn}(1)$[W]	29.79	31.41	33.71	38.01
$P^{alt}_{stat}(1)$[W]	13.99	13.74	12.43	9.96
RMSD (8)	1.29	1.32	1.42	1.03

Using the parameters from Table 5 and the function from Equ. (5), Figure 13 compares the modeled power consumption with the measured power values. The minimum and maximum error bounds are shown as dotted lines. Figure 13 shows that the modeling based on function (5) is quite accurate, i.e., Equ. (5) is well suited to capture the power consumption behavior. Figure 13 also depicts the model with the alternative power function (8). It can be seen that this modeling is less accurate especially for larger frequencies. This fact is also represented by the larger RMSD values in Table 5.

5.2. Time modeling

As described in Section 3.2, the execution time of the different versions of the RK methods decreases with increasing frequency. The parallel execution time using p threads for an ODE system of size n can be modeled by

$$T^V(n, s, p) = T^V_{basis}(n, 1, p) + s \cdot T^V_{scal}(n, 1, p) \tag{9}$$

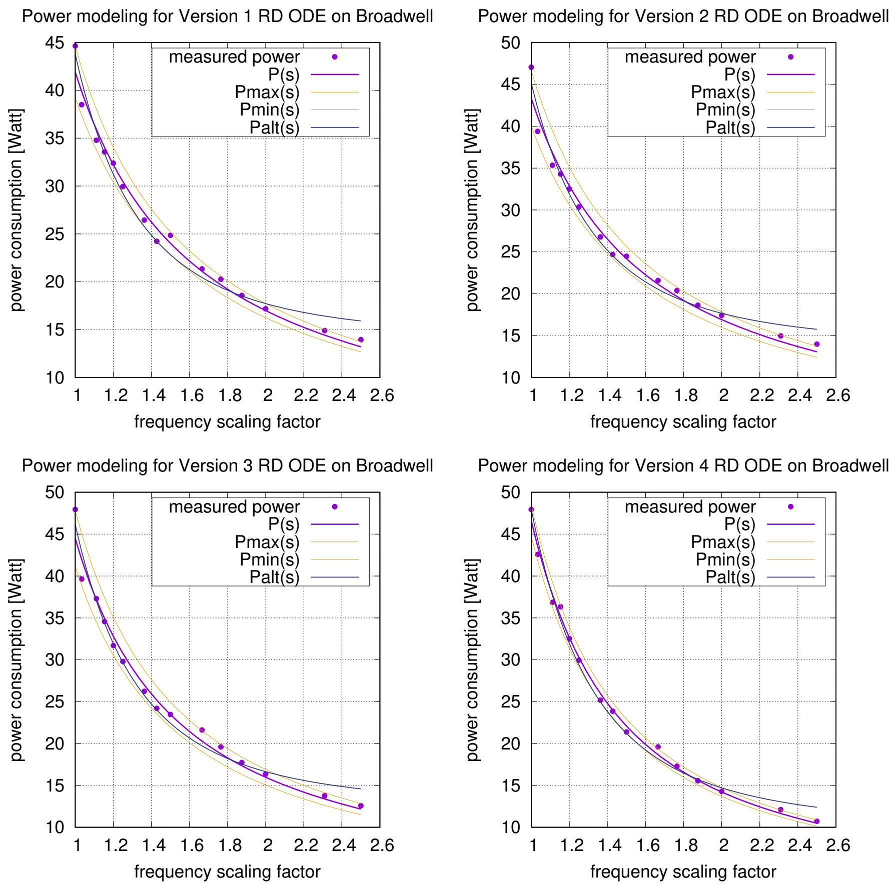


Figure 13: Modeling of the power consumption for the different RK Versions using Equ. (5) with error bounds given by Pmin(s) and Pmax(s) on the Broadwell processor.

where $T^V_{basis}(n, 1, p)$ represents the part of the execution time of version V that does not scale with the scaling factor s and $T^V_{scal}(n, 1, p)$ represents the part that scales with s and uses $s = 1$. Comparing Equ. (9) with Equ. (7), it is $T^V_{basis}(n, 1, p) = T^V_{mem}(n, p)$ and $s \cdot T^V_{scal}(n, 1, p) = T^V_{cache}(n, s, p) + T^V_{op}(n, s, p)$. Using the modeling with gnufit from Subsection 5.1 based on the measured execution times, the parameter values in Table 6 are obtained. The resulting modeling according to Equ. (9) is shown in Fig. 14. The figure shows that the resulting modeling is quite accurate. Moreover, the modeled execution times for the different RK versions correspond to the observations in Sections 4.3 and 4.5: the modeling for Version 4 leads to a significantly smaller value for $T_{basis}(n, 1)$, which can be explained by the smaller data volume to the main memory reported in Section 4.5. The required access time does not scale with s, since the main memory is not affected by DVFS. Similarly, due to the intensive use of the L2 cache reported in Section 4.5, see Table 4, the value for $T_{scal}(n, 1)$ of Version 4 is much larger than for the other versions. Overall, Equ. (9) is well suited to describe the

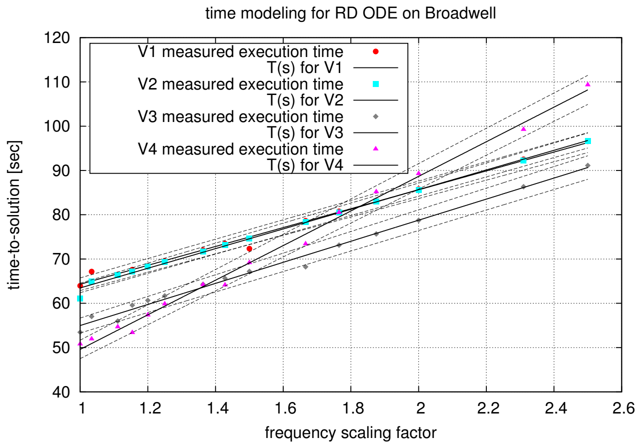


Figure 14: Modeling of the execution time with frequency scaling for RK Versions 1-4 using Equ.(9) with error bounds given by fmin(s) and fmax(s).

performance behavior of the different RK versions using frequency scaling.

Table 6: Modeling the execution time of the RD ODE on the Broadwell processor

Version	Version 1	Version 2	Version 3	Version 4
$T_{basis}(n, 1)$[sec]	42.98	41.26	31.19	10.51
$T_{scal}(n, 1)$[sec]	21.33	22.20	23.79	39.08
RMSD (5)	0.95	0.72	1.13	1.39

5.3. Energy modeling

The energy consumption of the different RK versions results from a combination of the power consumption and the execution time, see Equ. (6). Accordingly, the modeling of the energy consumption using frequency scaling uses the product of the power consumption model (Equ. (5)) and the time model (Equ. (9)) to obtain an estimation of the energy consumption. The result of this energy modeling is shown in Figure 15. It can be seen that the modeled energy values fit the measured energy values quite well. The maximum and average percentage deviations for the different RK versions over all operational frequencies are shown in Table 7. The average deviation between the modeled and the measured energy consumption is below 3 % and the maximum percentage deviation is 6.3 %. Thus, the modeling is quite accurate and results in a small percentage difference between the measured and the modeled energy values. Overall, it can be concluded that the model is well suited to capture the energy behavior of the different RK versions. In particular, the same model can be used for all RK versions and the model captures the differences between the different RK versions quite well and shows the advantage of Version 3 and 4 over Versions 1 and 2.

Table 7: Percentage deviation resulting for the energy modeling

Version	Version 1	Version 2	Version 3	Version 4
maximum deviation[%]	5.6 %	6.3 %	4.6 %	5.1 %
average deviation[%]	2.5 %	2.8 %	2.5 %	2.0 %

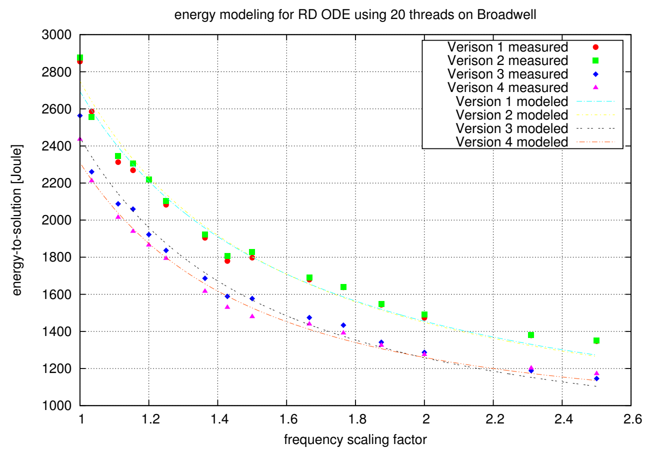


Figure 15: Modeling of the energy consumption with DVFS for the different RK Versions.

6. Related work

Runge-Kutta (RK) methods are popular solution methods for ODEs and implementations of RK methods are provided by several numerical libraries. Especially the implementation of DOPRI5 provided by Hairer and Wanner [20] is often used in practice. However, this code is specialized for DOPRI5 with fixed coefficients and does not support parallelism. Sequential implementations of several RK methods including DOPRI5 are also provided by RKSUITE [4], Matlab, and IMSL. Loop transformations to improve the locality behavior of sequential ODE solvers have been described in [32] covering sequential executions only. Analysis techniques for programs from scientific computing have been considered in [26, 27]. Some explicit RK methods are included as timesteppers in the PETSc library [2]. The default method is a 3rd order Bogacki-Shampine method which provides a 2nd order embedded solution. A 5th order Dormand-Prince method with a 4th order embedded solution is also supported. The parallelization approach of PETSc uses the MPI library, and a parallelization based on multithreading is not supported.

Approaches to improve the potential of parallelism in RK methods are often based on the development of new numerical algorithms with independent stage vector computations [46, 5, 6, 47, 45, 44, 31]. For most of these new methods, the parallel execution is based on the computation of a fixed number of stage vectors by different execution units. Thus, the parallelism is limited by the number of stage vectors to be computed in one time step, which is usually quite small. In contrast, this article considers classical RK methods with all their benefits concerning numerical accuracy and stability.

Moreover, the degree of parallelism is limited only by the size of the ODE system to be solved, which can be quite large, e.g., if ODEs resulting from discretized PDEs are considered.

The energy consumption of processors has been increasingly gained interest during the last years [38, 13]. To improve the energy consumption, several power-management techniques have been developed and used, including power capping [19], clock gating [3], and DVFS [42]. These techniques have been integrated in modern processor architectures [11]. Fundamental aspects of speed scaling and power-down scheduling are described in [9]. The present article considers the DVFS technique, which trades off performance for power consumption by lowering the operational frequency. Approaches to determine the voltage scaling factor that minimizes the total CPU energy consumption by taking both the dynamic power and the leakage power into consideration has been discussed in [23, 48] for sequential executions. Power models for multicore designs are developed in [13]. Power models for large-scale systems are described in [29, 28]. A detailed analysis of the power consumption of application codes and a power estimation method are given in [21]. A detailed discussion and usage of energy predictive modelling using performance monitoring counters is given in [40, 14].

Applying program transformations to ODE solvers and investigating the effect on the resulting execution time and energy consumption has already been pursued in [33]. The present article is an extension of this work, covering more hardware systems and performing a more detailed analysis of the performance and energy behavior. It also adds an investigation of the influence of the power consumption, see [34] for preliminary results, as well as a modeling of the power and energy consumption using the model from Section 5. Such a modeling has already been attempted in [35].

Energy consumption issues have also been considered in the context of specific applications, especially from the area of numerical analysis. The investigations include solvers for dense and sparse linear systems such as Cholesky factorization [1], preconditioned Conjugate Gradient methods [8], as well as iterative refinement techniques and reduced-precision computing features in modern accelerators [18]. A detailed survey of power and energy efficient techniques for (dense and sparse) linear algebra operations is given in [41]. The interplay between energy efficiency and performance for the numerical solution of PDEs is investigated in [17] with an emphasis on low-power ARM systems. The redesign of a hydrodynamic application towards energy efficiency has been investigated in [10] for a CPU-GPU combination. The reduction of the energy cost of applications from scientific computing is considered in [25] with an emphasis on systems of extreme scale, addressing both hardware and algorithmic efforts. A DVFS-based online frequency selecting algorithm for parallel iterative asynchronous methods running over grids has been presented in [15].

7. Conclusions

In this article, we have investigated how the performance and energy consumption of parallel ODE solvers can be influenced by applying application-specific program transformations. In particular, we have investigated the behavior of the resulting program versions for different numbers number of threads and different operational frequencies used for the execution. The investigation shows that the program transformations can have a significant effect on the performance and energy consumption. Depending on the specific execution platform, the execution time and the energy consumption can be reduced considerably compared to the basic version. The differences in the improvements of execution time and energy consumption, respectively, arise from the varying power drawing caused by the implementation versions examined.

The application-specific program transformations used to derive the series of different implementation versions of the ODE solvers are difficult to find by current compilers, since intermediate steps in the intertwined loop structures are required to exchange function evaluations with RK computations. In our investigations, this is indicated by using the highest compiler optimization level O3 for all implementation versions. If the compiler would be able to detect the transformations, this would have resulted in the same execution time for all implementation versions. Especially ODE solver Version 3 and 4 would not gain any performance improvement. However, as our measurements have shown, the performance and energy improvements are clearly visible for those two ODE solver versions. Our results may prepare the ground for a tuning methodology for ODE solvers, which enables a better performance or energy consumption by a switching between the different implementation versions for different situations, e.g., the specific ODE system to be solved, the size of the ODE system, the number of threads available, and the specific hardware capabilities.

An analytical modeling can help to estimate the performance and energy consumption of different versions. We have shown that it is possible to build such analytical models for performance and power consumption for the different implementation versions, which can be considered as a first step in the direction of an application-specific control of frequency scaling. Moreover, the models can be used for an a priori selection of a suitable implementation version, depending on the specific execution situation.

Application-specific program transformations as they have been investigated for embedded RK methods can also be derived for other ODE solvers such as iterated RK methods [30] or parallel Adams methods [45]. To do so, the computational structure of these methods has to be analyzed carefully to find the synchronization points required for the numerical correctness as it has been done for the embedded RK methods in this article.

Other numerical methods such as solution methods for PDEs could also benefit from application-specific transformations as they have been applied in this article. However, new transformations would have to be derived that are adapted to the computational structure of the specific numerical method. The transformations should address the computational kernels that are responsible for the main computational work. When deriving such transformations, the principal goal should be to obtain a good temporal access locality to the data structures used for the numerical values, i.e., after having accessed or computed a data element, this element should be used as often as possible. This approach can help to get a more efficient utilization of the cache memories of the processors.

Acknowledgment

This work is supported by the German Ministry of Science and Education (BMBF), project *Self-Adaptation of Time-step-based Simulation Techniques on Heterogeneous HPC Systems* (SeASiTe) under project number 01IH16012A/B. Moreover, we thank the LRZ Munich for the access to the Xeon system used for the experimental evaluation.

References

[1] J.I. Aliaga, H. Anzt, M. Castillo, J. C. Fernández, G. León, J. Pérez, and E. S. Quintana-Ortí. Unveiling the Performance-energy Trade-off in Iterative Linear System Solvers for Multithreaded Processors. *Concurr. Comput. : Pract. Exper.*, 27(4):885–904, March 2015.

[2] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.8, Argonne National Laboratory, 2017.

[3] E. Bezati, S. Casale-Brunet, M. Mattavelli, and J. W. Janneck. Clock-gating of streaming applications for energy efficient implementations on fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(4):699–703, 2017.

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

[5] K. Burrage. Parallel methods for initial value problems. *Applied Numerical Mathematics*, 11:5–25, 1993.

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

[7] J.A. Butts and G.S. Sohi. A static power model for architects. In *In Proc. of the 33rd Int. Symp. on Microarchitecture (MICRO-33)*, 2000.

[8] S. Catalán, F. D. Igual, R. Mayo, R. Rodríguez-Sánchez, and E. S. Quintana-Ortí. Time and Energy Modeling of a High-performance Multi-threaded Cholesky Factorization. *J. Supercomput.*, 73(1):139–151, January 2017.

[9] M. Chrobak. Algorithmic Aspects of Energy-Efficient Computing. In I. Ahmad and S. Ranka, editors, *Handbook of Energy-Aware and Green Computing*, pages 311–329. CRC Press, 2012.

[10] T. Dong, V. Dobrev, T. V. Kolev, R. N. Rieben, S. Tomov, and J. Dongarra. A Step towards Energy Efficient Computing: Redesigning a Hydrodynamic Application on CPU-GPU. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 972–981, 2014.

[11] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro*, 37(2):52–62, 2017.

[12] 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.

[13] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, and D. Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, February 2013.

[14] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky. Accurate Energy Modelling of Hybrid Parallel Applications on Modern Heterogeneous Computing Platforms Using System-Level Measurements. *IEEE Access*, 8:93793–93829, 2020.

[15] A. Fanfakh, J.-C. Charr, R. Couturier, and A. Giersch. Energy Consumption Reduction for Asynchronous Message Passing Applications. *The Journal of Supercomputing*, 73, 06 2017.

[16] C.W. Gear. Massive Parallelism across Space in ODEs. *Applied Numerical Mathematics*, 11:27–43, 1993.

[17] D. Göddeke, D. Komatitsch, M. Geveler, D. Ribbrock, N. Rajovic, N. Puzovic, and A. Ramirez. Energy efficiency vs. performance of the numerical solution of PDEs: an application study on a low–power ARM–based cluster. *Journal of Computational Physics*, (237):132–150, March 2013. DOI 10.1016/j.jcp.2012.11.031.

[18] A. Haidar, A. Abdelfattah, M. Zounon, P. Wu, S. Pranesh, S. Tomov, and J. Dongarra. The Design of Fast and Energy-Efficient Linear Solvers: On the Potential of Half-Precision Arithmetic and Iterative Refinement Techniques. In *Computational Science – ICCS 2018*, pages 586–600. Springer, 2018.

[19] A. Haidar, H. Jagode, P. Vaccaro, A. YarKhan, S. Tomov, and J. Dongarra. Investigating power capping toward energy-efficient scientific applications. *Concurrency and Computation: Practice and Experience*, 31(6):e4485, 2019. e4485 cpe.4485.

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

[21] J. Haj-Yihia, A. Yasin, Y. B. Asher, and A. Mendelson. Fine-Grain Power Breakdown of Modern Out-of-Order Cores and Its Implications on Skylake-Based Systems. *ACM Trans. Archit. Code Optim.*, 13(4), December 2016.

[22] Intel. *Intel 64 and IA-32 Architecture Software Developer's Manual, System Programming Guide*, 2011.

[23] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 275–280. ACM, 2004.

[24] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan & Claypool Publishers, 2008.

[25] J. Krueger, C. Rowen, D. Donofrio, J. Shalf, L. Oliker, M. Mohiyuddin, S. Kamil, and M. F. Wehner. Energy-Efficient Computing for Extreme-Scale Science. *Computer*, 42:62–71, 2009.

[26] M. Kühnemann, T. Rauber, and G. Rünger. A Source Code Analyzer for Performance Prediction. In *Proc. of 18th IPDPS, Workshop on Massively Parallel Processing (CDROM)*. IEEE, 2004.

[27] M. Kühnemann, T. Rauber, and G. Rünger. Performance Modelling for Task-Parallel Programs. In M. Gerndt, V. Getov, A. Hoisie, A. Malony, and B. Miller, editors, *Performance Analysis and Grid Computing*, pages 77–91. Kluwer, 2004.

[28] J. Mair, Z. Huang, and D. Eyers. Manila: Using a densely populated PMC-space for power modelling within large-scale systems. *Parallel Computing*, 2018.

[29] J. Mair, Z. Huang, D. Eyers, L. Cupertino, G. Da Costa, J.-M. Pierson, and H. Hlavacs. Power Modeling. In Jean-Marc Pierson, editor, *Large Scale Distributed Systems and Energy Efficiency: A holistic view*, pages 131–158. John Wiley and Sons, 2015.

[30] T. Rauber and G. Rünger. Parallel Iterated Runge–Kutta Methods and Applications. *International Journal of Supercomputer Applications*, 10(1):62–90, 1996.

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

[32] T. Rauber and G. Rünger. Optimizing Locality for ODE Solvers. In *Proc. of the 15th ACM Int. Conf. on Supercomputing*, pages 123–132. ACM Press, 2001.

[33] T. Rauber and G. Rünger. Energy and Performance Improvement of Parallel ODE Solvers by Application-specific Program Transformations. In *Proceedings of the 19th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-18), IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 967–976. IEEE, May 2018.

[34] T. Rauber and G. Rünger. How do loop transformations affect the energy consumption of Runge-Kutta methods? In *Proceedings of the 26th Euromicro International Conference on Parallel, Distributed, and Network-based Processing (PDP 2018)*, pages 499–507. IEEE, March 2018.

[35] T. Rauber and G. Rünger. DVFS RK: Performance and Energy Modeling of Frequency-Scaled Multithreaded Runge-Kutta Methods. In *Proceedings of the 27th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP 2019)*, pages 392–399. IEEE, February 2019.

[36] T. Rauber, G. Rünger, M. Schwind, H. Xu, and S. Melzner. Energy Measurement, Modeling, and Prediction for Processors with Frequency Scaling. *The Journal of Supercomputing*, 2014.

[37] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2):20–27, 2012.

[38] E. Saxe. Power-efficient software. *Commun. ACM*, 53(2):44–48, 2010.

[39] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, and D. Hackenberg. Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance. In *17th International Conference on High Performance Computing & Simulation, HPCS 2019, Dublin, Ireland, July 15-19, 2019*, pages 399–406. IEEE, 2019.

[40] A. Shahid, M. Fahad, R. R. Manumachu, and A. Lastovetsky. A Comparative Study of Techniques for Energy Predictive Modeling Using Performance Monitoring Counters on Modern Multicore CPUs. *IEEE Access*, 8:143306–143332, 2020.

[41] L. Tan, S. Kothapalli, L. Chen, O. Hussaini, R. Bissiri, and Z. Chen. A survey of power and energy efficient techniques for high performance numerical linear algebra operations. *Parallel Computing*, 40:559–573, 2014.

[42] K. M. Tarplee, R. Friese, A. A. Maciejewski, H. J. Siegel, and E. K. P. Chong. Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1633–1646, 2016.

[43] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *39th International Conference on Parallel Processing Workshops*, ICPP '10, pages 207–216. IEEE Computer Society, 2010.

[44] P. van der Houwen and B. Sommeijer. Parallel Iteration of high–order Runge–Kutta Methods with stepsize control. *J. of Comp. and Appl. Math.*, 29:111–127, 1990.

[45] P.J. van der Houwen and E. Messina. Parallel Adams Methods. *J. of Comp. and App. Mathematics*, 101:153–165, 1999.

[46] P.J. van der Houwen and B.P. Sommeijer. Parallel Iteration of high–order Runge–Kutta Methods with stepsize control. *Journal of Computational and Applied Mathematics*, 29:111–127, 1990.

[47] P.J. van der Houwen and P.B. Sommeijer. Parallel ODE Solvers. In *Proc. of the ACM Int. Conf. on Supercomputing*, pages 71–81. ACM, 1990.

[48] J. Zhuo and C. Chakrabarti. Energy-efficient dynamic task scheduling algorithms for DVS systems. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–25, 2008.