Noname manuscript No.
(will be inserted by the editor)

On the Potential of Significance-Driven Execution for Energy-Aware HPC

Philipp Gschwandtner · Charalampos Chalios · Dimitrios S. Nikolopoulos · Hans Vandierendonck · Thomas Fahringer

the date of receipt and acceptance should be inserted later

Abstract Dynamic Voltage and Frequency Scaling (DVFS) exhibits fundamental limitations as a method to reduce energy consumption in computing systems. In the HPC domain, where performance is of highest priority and codes are heavily optimized to minimize idle time, DVFS has limited opportunity to achieve substantial energy savings. This paper explores if operating processors Near the transistor Threshold Voltage (NTV) is a better alternative to DVFS for breaking the power wall in HPC. NTV presents challenges, since it compromises both performance and reliability to reduce power consumption. We present a first of its kind study of a *significance-driven execution paradigm* that selectively uses NTV and algorithmic error tolerance to reduce energy consumption in performance-constrained HPC environments. Using an iterative algorithm as a use case, we present an adaptive execution scheme that switches between near-threshold execution on many cores and above-threshold execution on one core, as the computational significance of iterations in the algorithm evolves over time. Using this scheme on state-of-the-art hardware, we demonstrate energy savings ranging between 35% to 67%, while compromising neither correctness nor performance.

Keywords significance · energy · unreliability · near-threshold voltage · fault tolerance

P. Gschwandtner · T. Fahringer
Technikerstrasse 21a, 6020 Innsbruck, Austria
E-mail: {philipp,tf}@dps.uibk.ac.at

C. Chalios · D. S. Nikolopoulos · H. Vandierendonck at Bernard Crossland Building, 18 Malone Road, Belfast, BT9 5BN, United Kingdom E-mail: {cchalios01,d.nikolopoulos,h.vandierendonck}@qub.ac.uk

1 Introduction

The interest in energy consumption in HPC has increased over the past decade. While high performance is still the main objective, many considerations have been raised recently that require including power and energy consumption of hardware and software in the evaluation of new methods and technologies. The motivational reasons are diverse, ranging from infrastructural limits such as the 20 MW power limit proposed by the US Department of Energy to financial or environmental concerns [19].

As a result, hardware industry has shifted their focus to include power and energy minimization into their designs. The results of these efforts are evident by features such as DVFS or power and clock gating. DVFS has been an efficient tool to reduce power consumption, however increased voltage margins –resulting from shrinking transistors– put a limit on voltage scaling. Although an energy-optimal voltage setting would often be below the nominal transistor threshold voltage, this would give rise to increased variation, timing errors and performance degradation that would be unacceptable for HPC applications. An alternative approach is to operate hardware slightly above the threshold voltage (also called near-threshold voltage, NTV). NTV would achieve substantial gains in power consumption but with acceptable performance degradation, which is caused by a rather modest frequency reduction and can be compensated by parallelism [12] .

Methods for tolerating errors in hardware have been studied in the past [6,9]. This resulted in several solutions at different levels of the design stack, in both hardware and software. However, these solutions often come with non-negligible performance and energy penalties. As an alternative, the shift to an approximate computing –also known as significance-based computing– paradigm has been recently proposed [3,12,13,18]. Approximate computing tries to trade reliability for energy consumption. It allows components to operate in an unreliable state by aggressive voltage scaling, assuming that software can cope with the timing errors that will occur with higher probability. The objective is to reduce energy consumption by using NTV and avoid the cost of fault-tolerant mechanisms.

In this work, we are trying to utilize the potential for power and energy reduction that NTV computing promises combined with significant-based computing. We investigate the effects of operating hardware outside its standard reliability specifications on iterative HPC codes, incurring both computational errors as well as reductions in energy consumption. We show that codes can be analyzed in terms of their significance, describing their susceptibility to faults with respect to their convergence behavior. Using the Jacobi method as an example, we show that there are iterative HPC codes that can naturally deal with many computational errors, at the cost of increased iterations to reach convergence. We also investigate scenarios where we distinguish between significant and insignificant parts of Jacobi and execute them selectively on reliable or unreliable hardware, respectively. We consider parts of the algorithm that are more resilient to errors as insignificant, whereas parts in which errors increase the execution time substantially are marked as significant. This distinction helps us to minimize the performance overhead due to errors and utilize NTV optimally.

We show that, in our platform, we can achieve 65% energy gains for a parallel version of Jacobi running at NTV compared to a serial version at super-threshold voltage along with time savings of 43%, when we execute with 20% of the super-threshold frequency.

Section 2 discusses related work relevant to our research. The notion of significance is introduced in Section 3. We will describe our methodology and experiment setup in Section 4 and analyze and illustrate our results in Section 5. Finally, Section 6 will conclude and provide an outlook for future research.

2 Related Work

Research in near-threshold voltage computing (NTC) has attracted considerable interest. NTC is a direct effect of industry's strive to keep up with Moore's law while coping with thermal and power limits. Prior research in NTC [4,12] investigates both hardware and software solutions to cope with the entailed proneness to faults and identify energy saving possibilities. However, this research either does not explore the effect of unreliability on unprotected codes or confines its exploration to probabilistic applications that can afford direct interventions to their convergence criterion and computation discarding [13]. Similarly, there are many works that explore the influence of errors on individual hardware units of processors, without further exporation of their implications on software [17].

Software methods for improving error resilience include checkpointing for failed tasks [9] or replication to identify silent data corruption [6]. Among others, Hoemmen and Heroux investigate iterative methods for their fault tolerance [8] and Elliott et al. quantify the error of single bit flips in progressing iterations of Jacobi [5]. However, these works do not investigate the impact of fault recovery on energy consumption or how fault resilience can be leveraged to reduce it.

Recently, there has been interest in exploiting approximate computing to build fault resilient systems. Leem et al. [13] build a system of few reliable and many unreliable cores. The system executes the control-intensive part of the application –which is highly fault intolerant– on reliable cores and the fault-tolerant compute-intensive part of the application on relaxed-reliability cores. This scheme achieves 90% or better accuracy of the output of applications even for 2×10^4 errors per second per core. Agarwal et al. [1], Misailovic et al. [14] and Rinard et al. [16] propose a static analysis-based technique to reduce the number of iterations in a loop without compromising correctness. In the same context, Baek et al. [3] provide a framework where the programmer specifies the functions and loops they want to approximate and the desired loss of Quality of Service (QoS). Then, the program is transformed to meet the QoS degradation target. Rinard et al. [15], propose to discard some tasks of the application and produce new computations that execute only a subset of the tasks of the original computation. Sampson et al. [18] propose a technique to distinguish the data types that need precise computation from the ones that can be approximated. They guarantee that the approximate instructions will never crash the program but only reduce power consumption.

3 Significance

We motivate the notion of code *significance*, i.e. that different parts of an application show different susceptibility to errors in terms of the change in the end result. This applies to data as well as operations and gives rise to considering partial protection methods, employed only where and when necessary. This distinction, coupled with the prospect of NTC, creates the opportunity for significant amounts of power savings by running

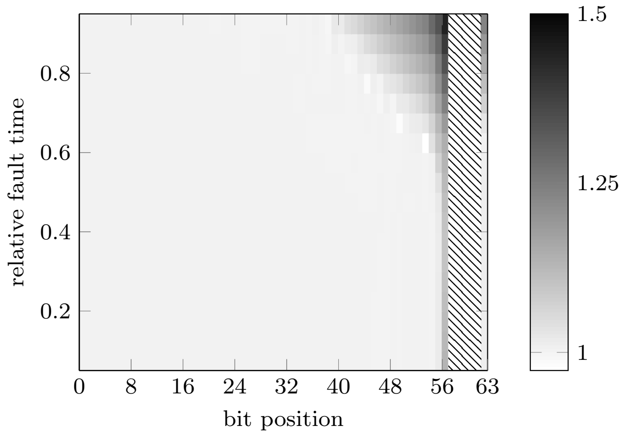

Fig. 1: Relative time overhead of Jacobi for faults in A at various iterations, averaged over all matrix positions, for all bit positions (N=1000). The hatched bar denotes divergence.

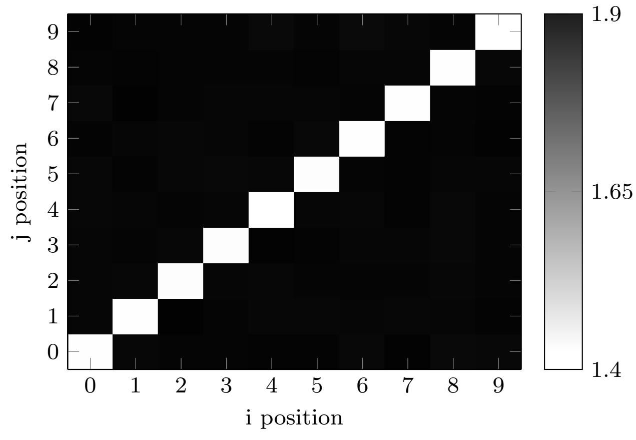

Fig. 2: Relative time overhead of Jacobi for faults in A at all matrix positions, averaged over all bit positions (N=10).

non-significant parts of the computation on unreliable hardware.

We want to illustrate the applicability of significance classification on iterative solvers and their resilience in the presence of faults. Iterative solvers operate on repeatedly updating the solution of a system of equations until it reaches the required level of accuracy. Errors occurring in these algorithms can be gracefully mitigated at the cost of an increased number of iterations to reach convergence. As a result, these applications are suitable candidates for trading accuracy for lower energy consumption.

We selected the weighted Jacobi method as a representative use case in order to study the resilience to errors in the broader class of iterative numerical applications. Jacobi solves the system $Ax = b$ for a diagonally dominant matrix A. It starts with an initial approximation of the solution, x^0, and in each step updates the estimation for the solution, according to

$$x_i^{(k+1)} = \omega \left(\frac{1}{a_{ii}} b_i - \sum_{j \neq i} a_{ij} x_j^k \right) + (1 - \omega) x_i^k. \tag{1}$$

The algorithm iterates until the convergence condition $\|Ax - b\| \leq limit$ is satisfied, and is guaranteed to converge if A is strictly diagonally dominant, i.e.

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|. \tag{2}$$

To demonstrate the applicability of significance to Jacobi, Figure 1 presents the effect of a single bit flip fault happening in matrix A at various iterations of an otherwise fault-free Jacobi run. It shows the relative overhead of Jacobi (i.e. the number of additional iterations) required to reach the convergence limit compared to a fault-free run. Generally, Jacobi exhibits a logarithmic convergence rate and later iterations are more significant due to the overhead required to recover from a fault. Furthermore, because the achieved residual for later iterations is lower than for earlier iterations (due to the better x that has been computed), later iterations also show higher sensitivity to faults. Both these factors render late Jacobi iterations more significant than early iterations [5]. This motivates the potential application of protection or recovery mechanisms for later Jacobi iterations only.

In addition, Figure 1 illustrates that Jacobi is able to cope well with flips happening in lower bit positions, as they cause little to no overhead. This can be attributed to the high precision of double-precision floating point numbers. Flips happening in the high bits of the exponent for elements of A however can have two possible outcomes, depending on their position and the error they introduce. If the flip causes a positive error in the floating point number and happens aside the diagonal, there is a risk of violating Jacobi's convergence condition of strict diagonal dominance for A, Eq. (2) (analogously for negative errors on the diagonal). These cases manifest themselves as the solid bar shape in Figure 1 for bits 57–62. For the majority of cases that violate this condition Jacobi does not converge and ends up with an residual of either *infinity* (*Inf*) or *not a number* (*NaN*), depending on the operations involved. Overall, Figure 1 shows that for most bit positions there is no protection or recovery necessary, except for a few high bits of the exponent that justify mitigation techniques.

In addition to Jacobi's varying significance depending on the progress of the algorithm, significance can also vary depending on the component that is exposed to a fault (A, b or x), as well as the position within a component. As an example, Figure 2 presents the relative overhead of injecting a fault on the diagonal and offside the diagonal of iteration matrix A (for illustrative clarity, we chose the problem size to be 10×10). The fact that elements on the diagonal show lower impact (and hence lower significance) can be attributed to the combination of two reasons. First, these elements are used in a division operation whereas the others are used in a multiplication (see Eq. (1)). Second, we use a uniform distribution to randomly initialize the elements of A and b, that leads to the majority of numbers being positive and greater than 1. Hence, a multiplication operation tends to increase any effect a fault might have, whereas a division generally reduces it. For this reason, our experiment results presented in Section 5 involving matrix A are based on sampling positions both on and aside the diagonal and computing a weighted average for the entire matrix. It should be noted that the weighting between diagonal vs. aside elements naturally changes with the matrix dimensions. As a consequence, the overall significance of A is also partially dependent on the input data size.

Detecting significance can be difficult for large codes. It is an algorithm-specific attribute and as a result input from programmers can be indispensible for a system that provides support for significance-based computing. Such a system would rely on the programmer's knowledge about the algorithm, who would denote e.g. which parts can be executed unreliably. Large pieces of software often compose from smaller mathematical kernels whose tolerance in faults has been extensively studied. Such studies could be used by programmers to mark code regions as significant or non-significant, without having to perform extensive profiling. Moreover, there is ongoing research for algorithmic detection of the significance of code based on profiling (e.g. automatic differentiation [20]). This approach studies the sensitivity of code blocks, monitoring the range of the output of a code block after perturbation applied to the input. A code block is then considered to be more sensitive to errors, the larger the range of possible output values is. Nevertheless, automatically and efficiently detecting code significance is still an open research area.

Furthermore, the design of a system for significance-based computing should provide fallbacks for applications that cannot afford unreliability in their execution. In order for these applications to be able to benefit from NTC, the system must employ software or hardware fault recovery mechanisms.

4 Methodology

This section describes the fault model of our work. Moreover, it elaborates the power and energy effects that we expect from operating hardware unreliably and provides details about the hardware and our measurement methods.

4.1 Fault Model

Following common practice in related work [17], faults can be categorized as:

- no impact: the fault has no effect on the application
- data corruption
  - silent: only detectable with knowledge about the application
  - non-silent: detectable without knowledge about the application
  - looping: faults that cause the application to loop
- other (e.g. illegal instructions, segmentation faults)

Of these fault classes we consider silent data corruption (SDC) faults since they are the most insidious in high performance computing. Signaling errors such as *Inf*, or *NaN* or application crashes due to illegal instructions are comparatively easily detectable. Also, looping might be identified by detecting fixed points in the iteration data of an application or constraints on the execution time of code regions. In contrast, SDCs can cause graceful exits with possibly wrong results, making them particularly important to be dealt with.

SDCs can be categorized further as persistent or non-persistent. Persistent faults occur at the source of the data in question, i.e if the data is read multiple times it will exhibit the same deviation each time. Non-persistent faults on the other hand are faults in temporary copies of data that are only used once (e.g. faults happening directly in execution units or registers).

We consider persistent faults, mappable to faults happening in CPU data caches that are read multiple times and might also be written back to main memory. We do not account for errors in machine code in instruction caches, because these can lead to non-recoverable errors. We assume that instruction caches are employed with protection mechanisms.

4.2 Energy Savings Through Unreliability

We explore execution schemes that deliberately compromise processor reliability, using NTV operation, for achieving power and energy savings in HPC codes [4]. Karpuzcu et al. [12] suggest that power savings between $10\times$ and $50\times$ are possible with NTV, albeit with a $5\times$ to $10\times$ reduction in clock frequency. Under these assumptions, a processor would consume $2\times$ to $5\times$ less energy per operation with NTV, compared to above-threshold voltage operation. Given a fixed power budget, a system design could replace few cores operating in the above-threshold region with many cores operating in the NTV region. A similar strategy of trading each reliable core with many unreliable NTV cores could be applied to achieve a fixed performance target.

4.3 Experiment Setup

The experimental testbed used for testing our method consists of an HPC node equipped with four Intel Xeon E5-4650 Sandy Bridge EP processors. Each CPU offers 8 cores with 32 KB and 256 KB of private caches each, and a processor-wide shared cache of 20 MB. The system runs a 3.5.0 Linux kernel and we used gcc 4.8.2 for compilation.

Our workload —a C implementation of Jacobi— was parallelized using OpenMP. The problem size $N = 1000$ was chosen such that the entire data resides in the last-level cache to minimize main memory interaction not covered in our energy measurement scope (described below), while still being large enough to ensure reasonable run times with regard to our measurements. Time measurements were done via x86's *rdtsc* instruction and we used Intel's well-documented RAPL interface to obtain energy consumption information [10]. Its PP0 domain provides readings encompassing all cores at a sampling rate of approximately 1 kHz and a resolution of 15.3 μJ, and recent work has shown it to be accurate enough for our use case [7]. To achieve consistent energy readings, the target hardware was warmed up for an ample amount of time before taking measurements.

Since our target hardware system only allows reliable operation, we have to simulate unreliable operation resulting in power and energy savings, as well as faults. The former can be achieved by correcting the energy consumption data obtained via RAPL with regard to the observations of near-threshold computing as discussed in Section 4.2. Moreover, to be able to simulate an arbitrary number of reliable or unreliable cores on non-configurable, commodity multi-core hardware, we need to take care when processing RAPL data as it includes off-core entities that might be oversized or not necessarily present in some cases (i.e. ring bus for a single core). To that end, we profiled the target CPUs with regard to their power consumption for all numbers of cores in a weak scaling experiment with Jacobi. Figure 3 shows the results of this endeavor. It indicates that the power consumption increases linearly with the number of cores with an offset for off-core entities of 7.1 Watts, which will be removed in subsequent data

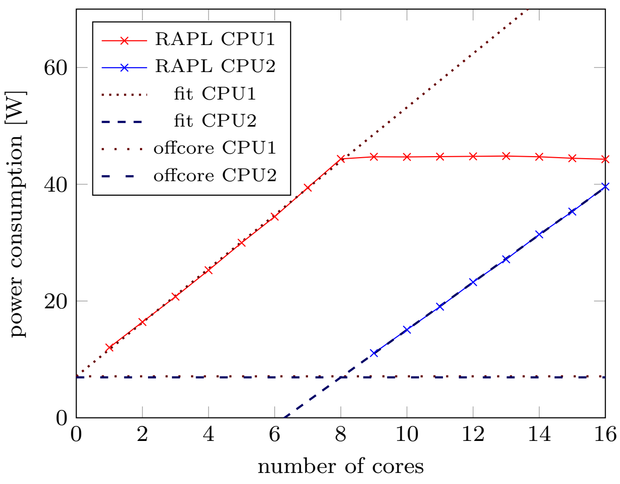

Fig. 3: Power consumption per number of cores on two Intel Xeon E5-4650 for a weakly scaling Jacobi run as measured by RAPL, and offcore amount as inferred via linear fitting.

analysis to provide a fair comparison between arbitrary numbers of reliable or unreliable cores. The figure also shows a different maximum power consumption for the two CPU samples (44.3 vs. 39.6 Watts), that can be explained by differences e.g. in supply voltage or the temperature.

As previously mentioned our target hardware only supports reliable operation, therefore we need to simulate faults in software. We inject persistent faults represented by bit flips in the original data at a range of bit positions of double precision floating point numbers prior to the computation of a Jacobi iteration. This simulates bit flips happening in the data caches of CPUs, that are accessed frequently during the computation. The implementation of the fault simulation is based on binary operators applied to the respective element in an inlined function, causing only negligible performance overhead compared to the overall execution time of a Jacobi iteration.

We assume a single bit flip per overall execution of Jacobi (i.e. over multiple iterations), since related work indicates that the effects of multiple faults will lead to similar observations [2] and because it reduces simulation complexity. Hence, an *experiment* is defined by

- the data component in which the fault occurs (in the case of Jacobi matrix A, or vectors x or b);
- the bit position i at which a flip occurrs;
- the Jacobi iteration k when the switch from unreliable to reliable mode occurs, with $1 < k < K$ and K denoting the total number of iterations and
- the Jacobi iteration j, before the execution of which the fault occurs, with $1 \leq j < k$.

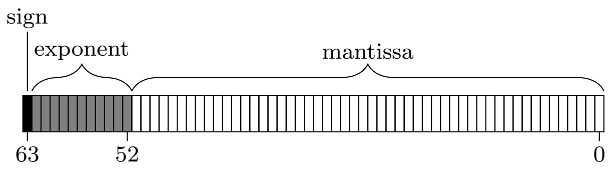

Fig. 4: IEEE 754 binary64 format

To minimize simulation time, we do not inject faults at every possible element of the vectors and matrices of Jacobi, but perform representative sampling (e.g. elements both on and aside the diagonal of a matrix) with respect to the algorithm. Furthermore we employ a convergence limit of a factor of 10. This means that we consider a faulty Jacobi run as not converging if it takes more than 10 times the number of iterations of a correct Jacobi run for the same input data set.

4.4 IEEE 754 Double-precision Floating-point Format

The data type in use for storing A, x and b in our Jacobi implementation is the default double-precision floating-point type of C, *double*. The analysis of our results in Section 5 is based on the bit positions within the IEEE 754 binary representation of these numbers, the *binary64* format. Elliot at al. already provide a detailed discussion regarding the effects of bit flips in this representation in [5]. Nevertheless, for clearness, we feel it is necessary to include a brief description of this representation and elaborate on the magnitude of errors introduced by bit flips, dependent on the position of the bit.

Figure 4 shows the binary layout of the widely-used *binary64* format. The first 52 bits (positions 0–51) correspond to the mantissa, the following 11 bits (positions 52–62) are used for the exponent and bit 63 denotes the sign. Furthermore, within the mantissa and the exponent, their lowest bits are the least significant ones. The decimal value v of a floating-point number is then computed by the formula

$$v = (-1)^s (1 + \sum_{i=0}^{51} b_i 2^{i-52} \times 2^{e-1023}), \tag{3}$$

where s denotes the sign bit, b_i the i-th bit of the mantissa and $e - 1023$ the exponent (stored with a bias of 1023). Hence, the altered floating point number resulting from a single bit flip in position j can be expressed as

$$a' = \begin{cases} a \pm 2^{j-52} \times 2^{e-1023} & \text{flip in mantissa,} \quad (4a) \\ a 2^{\pm 2^j} & \text{flip in exponent,} \quad (4b) \\ -a & \text{flip in sign.} \quad (4c) \end{cases}$$

We will evaluate the effect of these perturbations on the energy consumption and execution time of Jacobi in Section 5.

5 Results

In this section we compare executing Jacobi in parallel on unreliable hardware at near-threshold voltage (NTV) to sequential and parallel versions of Jacobi executed on reliable hardware at nominal voltage. Our results present three cases.

First, we run Jacobi at NTV in parallel throughout its entire execution (i.e. all iterations) and analyze and discuss the energy savings that can be gained compared to a sequential run at nominal voltage. Since we are dealing with an HPC code, we will also investigate the impact of operating at NTV on performance.

Second, the same analysis is repeated when comparing to a parallel execution of Jacobi.

Third, we explore the possibility of switching from NTV to nominal voltage for later iterations, motivated by our discussion in Section 3. We investigate whether later iterations of Jacobi are significant enough to justify the energy and performance expense. This could create a potential trade-off, since executing late iterations at nominal voltage also prevents late faults and thereby saves convergence overhead.

Given the absence of documentation on the clock frequency impairment that results from near-threshold computing, we consider two extreme cases: a frequency reduction by a factor of 5 (i.e. 20% of the nominal frequency, denoted by f=0.2, best case) and a reduction by a factor of 10 (i.e. 10% of the nominal frequency, denoted by f=0.1, worst case), as discussed in Section 4.2. We inject exactly one error in matrix A of Jacobi under NTV execution using the process outlined in Section 3.

The results illustrate the significance of matrix A, since it is the biggest component of Jacobi in terms of memory consumption and therefore presumably more prone to faults than smaller components. Furthermore, we assume the worst case regarding the iteration before which a fault can happen, i.e. the last unreliably executed one. All results presented are averages over 50 random input data sets for statistical soundness, with an overall variance of 10^{-5} for the relative overhead of the number of iterations for fault-injected Jacobi runs.

5.1 Sequential Reliable Jacobi

First, we investigate replacing a single, reliable core by multiple (in our case 16) unreliable cores to execute Jacobi under the same power envelope, as supported by

NTC (per-core power reductions of 10×–50×). Hence, we assume their maximum power consumption to be equal. Figure 5 illustrates the results of such a series of experiments, where in each experiment a fault happens in a different bit position. It shows the relative energy and time savings over a sequential, reliable run of Jacobi for all possible bit positions where faults may happen.

The results show that the effects of bit flip faults on energy and time may be categorized as follows:

A: no observable loss in energy or time,
B: observable loss in energy or time,
C: divergence.

Moreover, this classification coincides with the bit position that is flipped within an IEEE 754 double-precision floating-point number. Faults happening in bit positions 0–32 can be categorized as class A since they show no effect on energy savings. This can be contributed to both Jacobi's resilience to faults with small magnitudes, as well as the overall high precision of double-precision floating point numbers. Note that bits 0–32 are part of the mantissa and that a bit flip in these positions can affect normalized floating-point numbers by at most $2^{-20} \times 2^{e-1023}$ (see Equation (4a)). As a result, energy savings of 31% and 65% are possible for a 10x (f=0.1) and 5x (f=0.2) frequency reduction respectively.

Bit positions 33–54 and 63 are classified as B, with higher bit positions up to 54 showing a higher impact on energy and time. The maximum possible floating-point error for this class is $\pm 2^{-1}$ for the mantissa (c.f. Equation (4a), bit 51) and a factor of $2^{2^{54}}$ for the exponent (c.f. Equation (4b), bit 54). As such, energy savings are reduced to e.g. 48% for f=0.2 in the worst case. Bit 63 is not part of the exponent but holds the sign, and as such a flip in this position induces an absolute error of $2a$ (c.f. Equation (4c)) for a floating point number a, resulting in average energy savings of 52%. Class B warrants protection mechanisms if the user wishes to control the performance penalty incurred by NTV execution.

The missing data points at bit positions 55–62 are a member of class C since they are the highest significant bits of the exponent of a double-precision floating point number (induced errors between $2^{\pm 2^{55}}$ and $2^{\pm 2^{62}}$). For our setup, flips in any of these positions aside the diagonal of matrix A cause violations in Jacobi's convergence criterion, lead to divergence and have Jacobi break eventually with a non-silent *Inf* or *NaN* in most cases (see Section 3), which are easily detectable. Therefore, these bit positions should be protected in any case.

The correlation of time and energy savings in our results is a direct consequence of both our constant workload (i.e. arguably leading to constant power consumption) and the fact that we assume the same power budget for the unreliable cores and the reliable one.

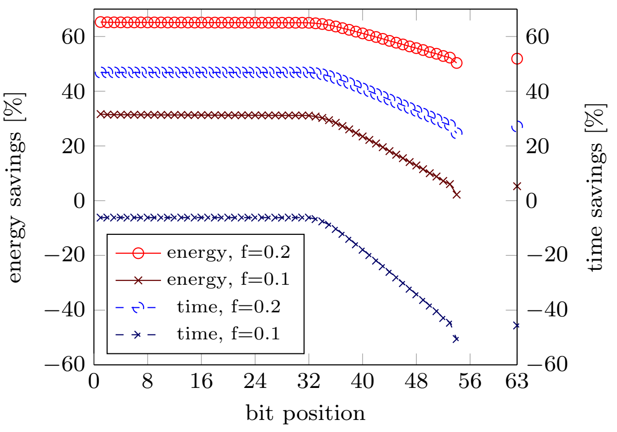

Fig. 5: Relative energy and time savings of an unreliable, parallel run of Jacobi on 16 cores compared to a reliable, sequential one. The missing data at bits 55–62 denotes divergence.

5.2 Parallel Reliable Jacobi

Our second experiment compares executing Jacobi in parallel at NTV against a parallel run at nominal voltage. The results of this comparison (Figure 6) show an identical classification of bit positions compared to our previous experiment. Nevertheless, one should note the lower performance compared to the previous scenario, attributed to the frequency reduction of near-threshold hardware by a factor of 5 to 10, as well as Jacobi's sub-linear scaling behavior. Therefore, for class A faults, performance losses between 413% and 925% are visible. Second, energy savings increase slightly (up to 35% and 67% respectively). This is expected due to the more energy-expensive nominal-voltage setup, since Jacobi does not scale perfectly with the number of cores. As a result, the increase in power consumption is not fully compensated by a reduction in run time, hence leading to a higher energy consumption. In turn, the relative energy savings of the NTV execution increase.

5.3 Significance-dependent Reliability Switching

In our third scenario, we investigate whether a fraction of the last iterations of Jacobi are significant enough to justify running them reliably at nominal voltage, and if so, when a switch from parallel execution at NTV to sequential execution at nominal voltage should occur. On one hand, switching to sequential execution increases run time and energy consumption. On the other hand, running at nominal voltage prevents faults and

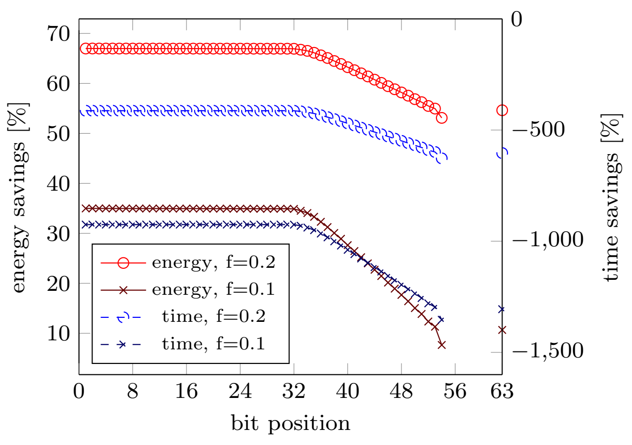

Fig. 6: Relative energy and time savings of an unreliable, parallel run of Jacobi compared to a reliable, parallel one on 16 cores. The missing data at bits 55–62 denotes divergence.

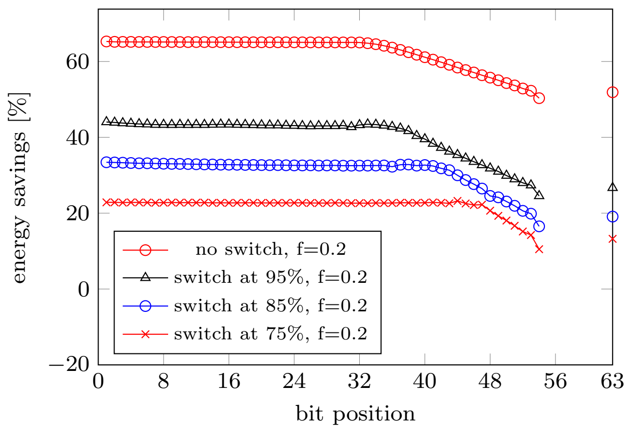

Fig. 7: Relative energy savings of an adaptive reliable/unreliable run of Jacobi on 16 cores compared to a reliable, sequential one, for switching to reliable hardware at 75%, 85% and 95% run time. The missing data at bits 55–62 denotes divergence.

guarantees convegence, without necessitating recovery iterations.

To that end, we run experiments where we switch from NTV execution to execution with nominal voltage at three points during a Jacobi run: 75%, 85% or 95% through completion. The intuition for this choice of switching points is Figure 1, where we observe that Jacobi experiences a significant slowdown when faults happen past the upper quartile of iterations. The energy consumption of these adaptive executions is depicted in Figure 7. For brevity, we only show results for f=0.2. Switching at a late point in time shows the highest savings for class A, as bit flips in this class have little to no effect on Jacobi and do not justify the energy expense of running at nominal voltage. Hence, while later Jacobi iterations are more significant, this increase in significance is too low to warrant protection. However, the switching point coupled with the increased significance of later iterations affects the classification of bit positions. Switching later implies that faults in lower bit positions will have higher impact since they happen in iterations with higher significance for convergence. As a result, switching at the 75% mark results in bit positions 0–47 to be categorized as class A, while the same class includes only bits 0–35 when switching at the 95% mark. This naturally changes the lower bit boundary of class B accordingly. However, it should be noted that it does not affect class C. If a bit flip in matrix A leads to divergence of Jacobi, it will always do so, regardless of when it happens. Overall, Figure 7 shows that switching at any of these three points in time does not pay off if the objective is to minimize energy consumption. The best strategy is to switch as late as possible (in our case at 95%), however all adaptive executions are outperformed by executing all iterations at NTV (65% energy savings vs. 43% for switching at 95%).

Figure 8 shows execution time with adaptive execution. Similar observations can be made for classification of flipped bits and their impact on energy consumption. However, the best strategy from an execution time perspective depends on the position of the bit flip, which affects the impact of late class B faults. For example, when a flip happens at bit position 48, switching at the 75% mark yields a relative time loss of 44%, while the time loss is 60% when switching at the 85% mark and 78% when switching at the 95% mark. Furthermore, it is evident that switching at the 85% mark or earlier already yields performance losses due to the time spent in sequential execution.

Overall, our results lead us to conclude that while Jacobi does indeed show an increase in significance for later iterations, this increase is generally too small — within the boundaries of our setup— to justify switching from parallel execution at NTV to sequential execution at nominal voltage.

6 Conclusion

In this work we explored the applicability and effect of near-threshold voltage (NTV) computation to a representative HPC code. We have shown that it can be a viable means of reducing the energy consumption, and that performance impairments caused by NTV can be mitigated via parallelism. We presented the notion of *significance-driven execution*, attributing varying significance to parts of a code or data and thereby deciding on whether they are a candidate for NTV computation or not. Our results show potential energy savings between 35% and 67%, depending on the use case. As such, significance-driven execution and NTV are a

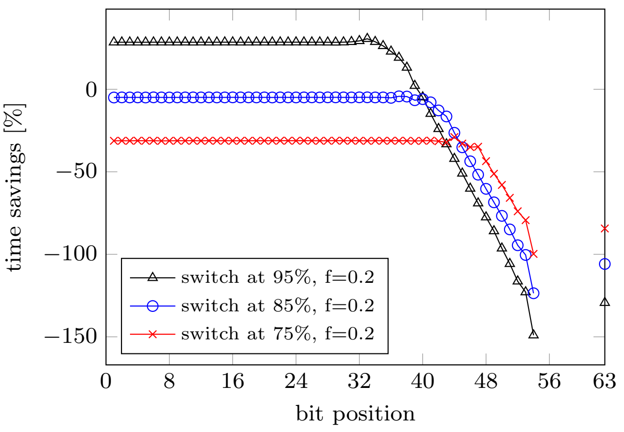

Fig. 8: Relative time savings of a hybrid reliable/unreliable run of Jacobi on 16 cores compared to a reliable, sequential one, for switching to reliable hardware at 75%, 85% and 95% run time. The missing data at bits 55–62 denotes divergence.

viable method of reducing the energy consumption in HPC environments without compromising correctness or performance. Future research opportunities include a detailed analysis of the effect of different degrees of parallelism, protection mechanisms for intolerable faults as identified in Section 5, and investigating and comparing the significance of additional iterative HPC codes. Additionally, ways of automatically determining the significance of code regions within compiler frameworks such as the Insieme compiler [11] could be explored.

Acknowledgements The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013), under grant agreements FP7-323872 (SCoRPiO) and 327744 (NovoSoft), the United Kingdom Engineering and Physical Sciences Research Council (EPSRC), under grants EP/L000055/1 (ALEA), EP/L004232/1 (ENPOWER) and EP/K017594/1 (GEMSCLAIM), and the Austrian Science Fund (FWF) under contract W1227-N16 (DK-plus CIM).

References

1. Agarwal, A., Rinard, M., Sidiroglou, S., Misailovic, S., Hoffmann, H.: Using code perforation to improve performance, reduce energy consumption, and respond to failures. Tech. rep., Massachusetts Institute of Technology (2009)
2. Ayatolahi, F., Sangchoolie, B., Johansson, R., Karlsson, J.: A study of the impact of single bit-flip and double bit-flip errors on program execution. In: Computer Safety, Reliability, and Security, pp. 265–276. Springer (2013)
3. Baek, W., Chilimbi, T.M.: Green: A framework for supporting energy-conscious programming using controlled approximation. SIGPLAN Not. **45**(6), 198–209 (2010)
4. Dreslinski, R.G., Wieckowski, M., Blaauw, D., Sylvester, D., Mudge, T.: Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. Proceedings of the IEEE **98**(2), 253–266 (2010)
5. Elliot, J., Müller, F., Stoyanov, M., Webster, C.: Quantifying the impact of single bit flips on floating point arithmetic. Tech. rep., Tech. Rep. ORNL/TM-2013/282, Oak Ridge National Laboratory, One Bethel Valley Road, Oak Ridge, TN, 2013. 6, 9 (2013)
6. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R.: Detection and correction of silent data corruption for large-scale high-performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 78. IEEE Computer Society Press (2012)
7. Hähnel, M., Döbel, B., Völp, M., Härtig, H.: Measuring energy consumption for short code paths using RAPL. SIGMETRICS Perform. Eval. Rev. **40**(3), 13–17 (2012)
8. Hoemmen, M., Heroux, M.: Fault-tolerant iterative methods via selective reliability. In: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society, vol. 3, p. 9 (2011)
9. Hursey, J., Squyres, J., Mattox, T., Lumsdaine, A.: The design and implementation of checkpoint/restart process fault tolerance for open mpi. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pp. 1–8. IEEE (2007)
10. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B Part 2 (2013)
11. Jordan, H., Thoman, P., Durillo, J., Gschwandtner, P., Fahringer, T.: A multi-objective auto-tuning framework for parallel codes. In: Supercomputing, 2012 Conference. IEEE (2012)
12. Karpuzcu, U., Kim, N.S., Torrellas, J.: Coping with parametric variation at near-threshold voltages. Micro, IEEE **33**(4), 6–14 (2013)
13. Leem, L., Cho, H., Bau, J., Jacobson, Q.A., Mitra, S.: Ersa: Error resilient system architecture for probabilistic applications. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010, pp. 1560–1565. IEEE (2010)
14. Misailovic, S., Sidiroglou, S., Hoffmann, H., Rinard, M.: Quality of service profiling. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pp. 25–34. ACM (2010)
15. Rinard, M.: Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In: Proceedings of the 20th annual international conference on Supercomputing, pp. 324–334. ACM (2006)
16. Rinard, M., Hoffmann, H., Misailovic, S., Sidiroglou, S.: Patterns and statistical analysis for understanding reduced resource computing. SIGPLAN Not. **45**(10), 806–821 (2010)
17. Saggese, G.P., Wang, N.J., Kalbarczyk, Z.T., Patel, S.J., Iyer, R.K.: An experimental study of soft errors in microprocessors. IEEE micro **25**(6), 30–39 (2005)
18. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: Enerj: Approximate data types for safe and general low-power computation. SIGPLAN Not. **46**(6), 164–174 (2011)
19. Tolentino, M., Cameron, K.W.: The optimist, the pessimist, and the global race to exascale in 20 megawatts. Computer **45**(1), 95–97 (2012)
20. Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., Wunsch, C.: Openad/f: A modular open-source tool for automatic differentiation of fortran codes. ACM Trans. Math. Softw. **34**(4), 18:1–18:36 (2008)