Dealing with Hardware Faults in Energy-Efficient Static Schedules of Multi-Variant Programs on Heterogeneous Platforms

Jörg Keller
Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
Joerg.Keller@FernUni-Hagen.de

Christoph Kessler
Dept. of Computer and Information Science (IDA)
Linköping University
58183 Linköping, Sweden
Christoph.Kessler@liu.se

***Abstract*—We investigate the energy-efficient execution of programs with a sequence of program parts, each part executable by multiple variants on different execution units. We study their behaviour under the presence of crash faults on a computing platform with heterogeneous execution units like multicore, GPU, and FPGA. To this end, we extend a static scheduling algorithm for computing the sequence of variants leading to minimum runtime, minimum energy consumption, or a weighted sum of both, to consider cases where one or more program variants cannot be used anymore from some execution point on, due to failure of the underlying execution unit(s). This extension combines the advantageous results of static scheduling, known in the fault-free case, with avoidance of overhead for re-scheduling in case of a fault. We evaluate our algorithm with synthetically generated progam task graphs. The results indicate that, compared to computing a new schedule for each fault case, our algorithm only needs 55% of the scheduling time for 8 variants.**

***Index Terms*—static scheduling, energy-efficient execution, fault-tolerant execution, optimization algorithm**

I. Introduction

Modern computer architectures are increasingly power constrained while the computational demands from the application side continue to grow. In order to deliver improved performance within a sustainable power envelope, the heterogeneous computing paradigm is getting widely adopted [1]. In fact, entire new application areas such as deep learning have appeared in the last decade thanks to the availability of affordable, GPU-powered heterogeneous computer systems. With radically new hardware technologies not ready yet to take over, we expect that the trend towards using heterogeneous architectures will grow even further.

At the same time, the steady density increase in silicon technology as postulated in Moore's Law leads to tiny structure sizes that are more prone to wear-out, and to higher power density which may result in temporary or even permanent loss of some hardware components e.g. due to damages from local overheating [2]. While certain hardware components are critical (such as the master core, a general purpose CPU responsible for coordinating a heterogeneous computation) and should be hardened against permanent faults by hardware mechanisms (such as a standby core), the loss of non-critical components can be exposed to the software. The classification of system components into critical and non-critical ones and the assumption of a reliable critical core are common in fault-tolerant systems design, see e.g. Sha [3]. Applications to be deployed on a heterogeneous architecture that have strict robustness requirements, such as driver assistance systems in modern cars, should be designed to be able to tolerate the loss of some non-critical hardware component (e.g., a certain accelerator) and remain operational, with possibly reduced quality of service or reduced performance. Permanent faults may not necessarily be caused by hardware damage only but could also occur for instance in heterogeneous systems equipped with hot-pluggable external accelerators, such as USB-connected external GPUs, which might be disconnected at any time. In real-time systems, a component that takes longer to execute a task than the worst-case execution time (WCET) analysis predicts may be classified as faulty, too.

In this paper we consider a static scheduling method for efficiently dealing with the (potentially) permanent loss of a non-critical execution component in a heterogeneous system when optimizing the device mapping for a sequence of computations. We assume that the computation is structured as a sequence of phases that each might have a number of code variants that each run on a different execution unit, accordingly with different impact on time and energy usage. We extend a known static scheduling algorithm to compute the sequence of phase variants with minimum runtime, energy consumption or a combination of both, to comprise also cases where one or several variants are not available anymore due to the fault of a hardware component. Thus, we carry over the superior scheduling results of static scheduling (compared to dynamic scheduling) from the fault-free case to the case of a fault, and at the same time avoid overhead at runtime due to computing a new schedule when a fault is detected. Beyond the ability to remain operational in case of a fault, computing phase sequences for all considered fault cases in advance allows to give guarantees on performance and/or energy consumption, and thus goes beyond computation of a next-best sequence in case of a fault.

We implement our scheduling algorithm as a sequential program and compare our implementation to the straightforward alternative: to compute additional schedules for each fault situation, i.e. the fault of any component during execution of any phase. The comparison is done with the help of synthetically generated phase sequences, where for each phase a number of variants with their associated runtimes and energy consumptions is computed according to a random distribution. Our experiments indicate that for 8 variants, our scheduling on average needs only 55% of the scheduling time of the alternative.

Our main contributions are the following:

- We extend a static scheduling method for programs modelled by sequences of phases with multiple code variants to cover faults of components in a heterogeneous computing system.
- We implement the extended static scheduler and its alternative.
- We evaluate the extended static scheduler with synthetically generated program structures of different sizes for heterogeneous computing systems of different sizes.

The remainder of this paper is organized as follows. In Section II, we present background information and related work. In Section III, we present the scheduling algorithm and its extension to cover fault situations. Section IV reports on the experiments performed to evaluate our proposal. Section V presents a conclusion and an outlook on future work.

II. Background and Related Work

A. Platform Model

We consider a heterogeneous computing system that comprises a multitude of components or execution units. We assume that there exists a powerful and fault-resistant processor core. This may either be a core specially hardened against failures, but it may also be a realized by using multiple cores in a (virtual) duplex or triple system [4]. Beyond, there may be multiple cores of the same core type, there may be processor cores of different computing power, and there may be accelerators of different type, such as a GPU usable for application programs, an FPGA or an Intel Xeon Phi.

All components may differ in their execution speed and in their power consumption. To compare the execution of two versions of a code on different components, we can use the runtime of the codes or the respective energy consumption, computed as the product of the runtime and the (average) power consumption of this component during that time. The components may have different speed-/power-states, i.e. they might use different operating frequencies, which result in different power consumption. We assume that the supply voltage is always set to the lowest value possible for the chosen frequency. For many chips, the minimum voltage now is identical for large frequency ranges, so that the potential savings by frequency scalings are limited. Furthermore, the number of useful operating frequencies often is rather small, because the growing importance of static power consumption diminishes the reduction in power consumption when the operating frequency is reduced, so that the reduced power consumption cannot make up for the increased runtime, so that runtime *and* energy consumption for a particular task increase. In [5], the energy-optimal frequency for big and LITTLE cores and several different types of tasks (i.e. different instruction mixes) was 1,200 MHz, i.e. quite high, so that only one of the few higher frequencies (1,400 MHz for LITTLE, 1,400 or 1,600 MHz for big) could be used to improve either runtime or energy consumption for executing a task. The components typically also support sleep states with low or even negligible power consumption.

The components may have separate memories, so that if a separate component shall continue to work on the intermediate result of another component, a data transfer might be necessary via a communication system (for simplicity, one may assume a system bus) that connects the components. Also such a data transfer incurs time and energy consumption. In the programs we consider (see next subsection) only one component is in use at any time. We assume that the other components are put to an energy-saving mode so that their contribution to the total power consumption can be neglected during that time. An exception is the central processor core, which always runs.

All components except the central processor core may be subject to a fault at any time. We assume a fail-stop model, i.e. a component that fails will be silent [6] and will not consume further energy. In similarity to [7] we assume a runtime system that is notified about faults, aborts on-going transfers, and restarts the actual phase computation on a different component if the failed component was in use at the time of the fault.

B. Application Model

We model a program as a sequence of n tasks or phases, where task 0 is the initialization, task $n-1$ is the deinitialization, and the tasks in between might be given e.g. as calls to a computational library. For example, Hansson and Kessler [8] model a solver application for ordinary differential equations based on libsolve [9], which calls *copy*, *absaxpy*, *axpy*, *scale* and *absquotMax* inside a loop, so that a multitude of phases exist. Beyond ODE integration, this sequence or iteration based program structure is characteristic for applications e.g. in solving systems of linear equations (such as Conjugate Gradient based solvers), time-discretized simulations (e.g., N-body simulations), online sensor data processing (e.g., camera image frame preprocessing) in control and surveillance applications, or convolutional neural network inference. We only target tasks with computational workloads, as I/O and/or lock-based synchronization requires different approaches for runtime and energy prediction. For each task, there may exist a multitude of code variants, i.e. implementations with different algorithms, different operating frequencies and/or different execution unit types to be found in a heterogeneous machine. Examples of code variants are: sequential execution on a single core of the CPU (switching all other cores to sleep mode), parallel execution on all cores of the CPU, parallel execution on the graphics processing unit (GPU), or parallel execution on any other accelerator, such as Intel Xeon Phi or FPGA. While, in principle, several variants could use the same execution resource, we assume in the following that each task has at most one variant for each execution resource, where variant 0 runs on the master core. This assumption is however no true restriction and only serves to simplify our notation by skipping one indirection level in indexing that maps the variants to the used execution resources. Hence, we will from now on use the terms *variant* and *execution resource/unit* interchangeably.

For each variant of each task, we can forecast (either by experiments or analytically) the runtime and the energy consumption of this variant, given the target machine configuration and the current execution context (such as the activity status of the used execution unit, its operating frequency and whether the operand data still needs be transferred or is already locally available).

We are aware that the runtime of a particular variant might not be deterministic for a variety of reasons: dependence on data distribution, noise, external influences (such as frequency reduction due to heat). So we might be forced to work with performance figures derived by averaging over several trials, or to assume worst case figures, depending on the optimization context. For example, the question of hard or soft real-time requirements may change the figures used.

A concrete execution on the chosen platform is characterized by the *sequence* $(v_0, \ldots, v_{n-1})$ of the variants chosen.

Please note that not all tasks might offer variants for all types of execution units. For example, the first and last tasks are normally executed on the master core of the CPU ($v_0 = v_{n-1} = 0$), and for some task a particular accelerator might not provide the computational capability to execute this task. We denote the subset of variants available for task i by V_i, i.e. $v_i \in V_i$.

C. Related Work

Kicherer and Karl [10] combine dynamic scheduling of tasks on a heterogeneous system for performance with fault-tolerance. Msadek et al. [11] propose an autonomous system to map services to nodes e.g. to improve load balancing, to map more important services on trustworthy nodes, and combine this with fault tolerance. In contrast, we target static scheduling and consider a multitude of targets for optimization.

Eitschberger [7] computes static schedules for tasks with dependencies on parallel machines with speed scaling and considers the trade-off between runtime, energy efficiency and fault tolerance. However, his target platforms are homogeneous, and he treats fault-free and fault cases separately.

Izosimov [12] considers transient faults, which are overcome by re-execution. In contrast, we consider permanent faults.

Hansson and Kessler [8] consider optimal selection of variants in programs modeled as sequences of multi-variant tasks, but do not consider fault tolerance.

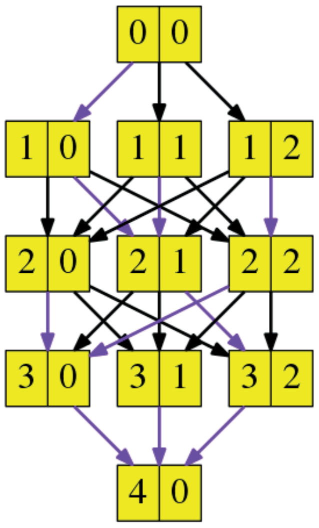

Fig. 1. Execution graph of program with $n = 5$ tasks (numbered 0,...,4, first entry in each box) and $p = 3$ variants per task (numbered 0,...,2, second entry in each box). Edge cost labels are omitted for better readability. Purple arrows show the shortest continuation path from each task variant to the sink $(4, 0)$. The overall shortest path from source to sink is here $((0, 0), (1, 0), (2, 1), (3, 2), (4, 0))$.

III. Scheduling Algorithm

A. Shortest Path Algorithm

All possible execution options, i.e. all possible sequences, can be expressed as a directed graph $G = (V, E)$, called *program execution variant graph*, where the set V of nodes is comprised of all variants of all tasks, i.e.

$$V = \{s_{i,j} \mid i = 0, \ldots, n-1, \quad j \in V_i\}$$

and the set of edges is comprised of the complete directed bipartite (sub)graphs between the nodes of successive tasks, i.e.

$$E = \{(s_{i,j}, s_{i+1,\tilde{j}}) \mid i = 0, \ldots, n-2, \quad j \in V_i, \quad \tilde{j} \in V_{i+1}\} .$$

Such an execution graph for $n = 5$ tasks and $p = 3$ variants per task (except for first and last task, see previous section) is illustrated in Fig. 1. A particular execution sequence $(v_0, \ldots, v_{n-1})$ is given by a path from $s_{0,0}$ to $s_{n-1,0}$, i.e. from source to sink, comprising all nodes s_{i,v_i}. Such an execution, choosing variants 0, 1 and 2 for the inner stages, is highlighted as a path in Fig. 1.

Each node, i.e. each variant of each phase, can be attributed with its runtime and energy consumption. In addition, when two successive tasks are realized in different variants, i.e. if $v_{i+1} \neq v_i$, and when the associated execution units use different memories, then a data transfer between those memories (e.g. between CPU main memory and GPU memory) or a conversion of data representation (e.g. when switching between word-parallel and bit-slice representation) might be necessary with associated runtime and energy consumption, attributed to the edge $(s_{i,v_i}, s_{i+1,v_{i+1}})$. We model this by defining transfer cost matrices $tr_i \in V_i \times V_{i+1}$, where $tr(i,j) = 0$ if variants v_i and v_j use the same memory and data representation.

In order to only deal with edge attributes (also called weights in the sequel), we add the node weights to each outgoing edge. This forces us to ignore the weight of the sink node. However, as this node is part of every execution sequence, it will not influence choice of paths with some optimality criteria. We denote the attributes by $t_{i,j,\tilde{j}}$ for the *runtime* of variant j in task i plus the transfer or conversion time when switching to variant $\tilde{j}$ in task $i+1$, and $e_{i,j,\tilde{j}}$ for the associated *energy* consumption. For a concrete execution sequence $(v_0, \ldots, v_{n-1})$, the total runtime $T(v_0, \ldots, v_{n-1})$ (and energy consumption $E(v_0, \ldots, v_{n-1})$) can be computed as the sum of the runtimes (energy consumption values) along the path from source to sink node:

$$T(v_0, \ldots, v_{n-1}) = \sum_{i=0}^{n-2} t_{i,v_i,v_{i+1}} \tag{1}$$

$$E(v_0, \ldots, v_{n-1}) = \sum_{i=0}^{n-2} e_{i,v_i,v_{i+1}} \tag{2}$$

The variants for a concrete execution might be selected prior to the actual execution, constituting a case of static scheduling. As scheduling is an optimization problem, a number of target functions may be suitable depending on the optimization goal, some examples being:

- minimizing total runtime,
- minimizing total energy consumption,
- minimizing the weighted sum of runtime and energy consumption,
- minimizing the energy consumption while the runtime does not exceed a given deadline.

In the present paper, we only consider the first three options. The choice of the variants thus becomes a variant of the single-source shortest path problem (SSSP) [8]. For the graph structure of our application scenario, solving this optimization problem is simpler than for a general graph. We construct the optimal path by starting in the sink node $s_{n-1,0}$ and we execute the shortest path algorithm by determining the minimum distances of the nodes in phase i from the given distances of the nodes in phase $i+1$, for $i = n-2, \ldots, 0$. This is a variant of Dijkstra's SSSP algorithm [13] but we can avoid the use of a priority queue. The time complexity is linear in the size of the graph ($O(\#\text{edges})$), i.e. $O(n \cdot p^2)$.

While we only need the shortest path from the source node to the sink node for the scheduling in the fault-free case, i.e. for computing the execution sequence $(v_0, \ldots, v_{n-1})$, solving this problem is not faster than computing the shortest path from all nodes to the sink. As a by-product, this simplifies to cover fault situations, which are treated in the next subsection.

B. Extending the Execution Graph

We extend the execution graph to cover fault cases in two steps.

In step 1, we attribute each edge $e = (u, v)$ which belongs to the shortest path from u to the sink with a set $N(e)$ that specifies all nodes on the remainder of that shortest path. We compute the attributes starting with the (hypothetical) edge leaving the sink node which is attributed with the empty set. For any other edge $e = (u, v)$, we define $N(e) = \{v\} \cup N(e')$ where e' is the shortest path edge that leaves v. We also attribute each node $s_{i,j}$ with the set $Ex(s_{i,j})$ of nodes **not** needed in the future when following the shortest path from that node to the sink node. The attribute can be computed by set exclusion, i.e. $Ex(s_{i,j}) = \{0, \ldots, p-1\} \setminus N(e)$ where e is the shortest path edge that leaves $s_{i,j}$.

In step 2, we model all fault situations to be tolerated by a set F. If only the failure of a single component must be tolerated, and each variant is executed on a different component, then $F = \{\emptyset, \{1\}, \ldots, \{p-1\}\}$. The empty set is included as it specifies the situation without a fault. Variant 0 executed on component 0 (the master core) is not included as it runs the sink node task and hence its failure cannot be tolerated. The maximum F, where up to $p-1$ component/variant faults can be tolerated, is the power set of $\{1, \ldots, p-1\}$.

We extend the graph by nodes $s_{i,j}^S$ for all $i = 0, \ldots, n-1$, $j \in V_i$ and $S \in F$. By $s_{i,j}^S$ we mean that task i is executed in variant j, and that fault status S is present, i.e. that the shortest path from this node to the sink node may not use a variant from S anymore. Please note that we allow that $j \in S$, which seems contradictory, but will be explained below. The original nodes $s_{i,j}$ correspond to nodes $s_{i,j}^{\emptyset}$, as the shortest path from $s_{i,j}$ to the sink node is allowed to use all variants.

For all S where $S \subseteq Ex(s_{i,j})$, we can identify node $s_{i,j}^S$ with $s_{i,j}$, as the shortest path from $s_{i,j}$ to the sink does not use any node from S. For the nodes $s_{i,j}^S$ that cannot be identified with an already existing node, we add outgoing edges: node $s_{i,j}^S$ is connected to all nodes $s_{i+1,j'}^S$ where $j' \notin S$. Thus, a node $s_{i,j}^S$ with $j \in S$ does not have incoming edges.

Now we can compute shortest paths from these nodes to the sink node. As the path at some place may converge with an existing path, if a follow-up node is identified with a previously existing node, the effort for this computation is reduced in contrast to using a different, modified graph for each fault situation.

Figure 2 illustrates the extended graph for the execution graph of Figure 1.

C. Scheduling in Fault Cases

In the fault-free case, we just use the computed execution sequence along the shortest path from source to sink node and execute the respective variants, one by one, doing data transfers in-between if necessary.

In the case of a fault, we use the extended graph in the following way. We assume a fail-stop model, where a component that fails is silent [6] for the rest of the execution, and where the runtime system learns or is notified about the failure [7] (e.g. by repeated probing of components.) Thus, at each point in time, we know the current fault status S, i.e. the set of failed components.

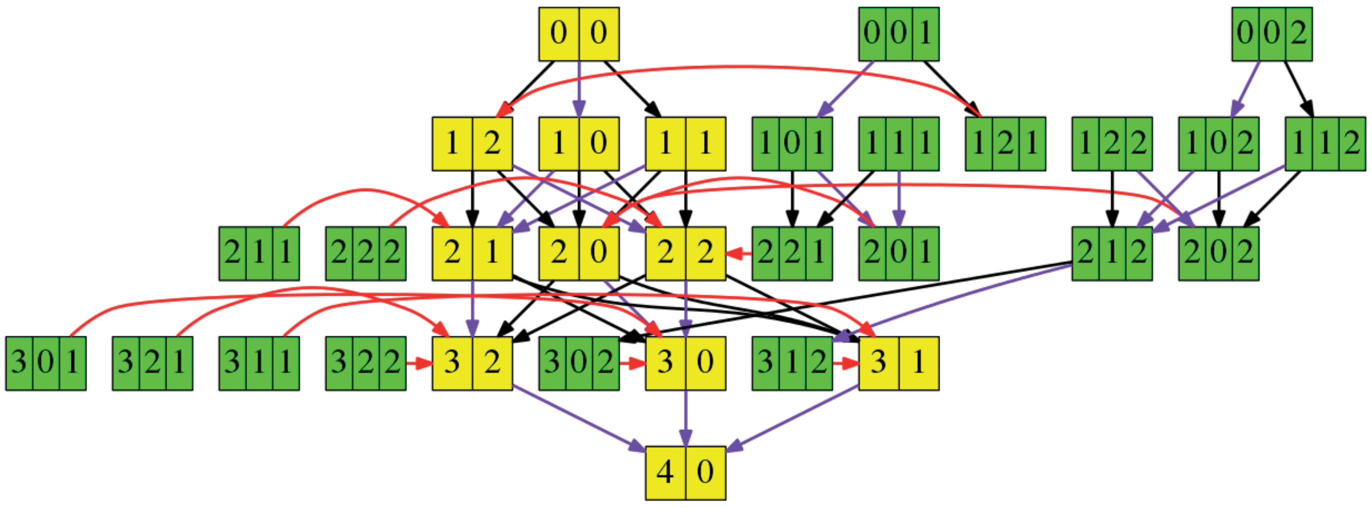

Fig. 2. Extended execution graph considering at most one faulty hardware unit (variant) for the program with $n = 5$ tasks of Figure 1. Variant 0 is assumed to be always fault-free (hardened by hardware mechanisms). The execution graph of Figure 1 is contained as a subgraph modeling the fault-free case (yellow nodes). The green nodes model all execution states in the case of one fault. They are identified by triplets (i, j, k) where $0 \leq i \leq n-1$ denotes the stage number, $0 \leq j < |V_i|$ denotes the variant of stage s, and $1 \leq k < |V_i|$ denotes the variant assumed to become unavailable during the execution of (i, j). Again, purple edges show the shortest continuation path to the sink from each node. Red edges indicate that node (i, j, k) can be identified with node (i, j), because the shortest path leaving (i, k) will not use variant k. For example, node $(1, 2, 1)$, assuming variant 1 being broken, can delegate to the shortest continuation path from regular node $(1, 2)$ which passes along $(2, 2)$, $(3, 0)$ and $(4, 0)$ and thus does not use the faulty component of variant 1.

If a component k fails, then we extend the fault status by including k into S, and check if the updated S still is a subset of F. If not, then the computation cannot be finished anymore. For the remainder, we focus on the case $S \subseteq F$.

If we currently execute task i in variant $j \neq k$, then we look up node $s^S_{i,j}$ in the graph and follow the shortest path from that node. If $k = j$, we must do a bit more, as the work already invested in the current task is lost. We look up the variant j' that has been used for task $i - 1$. Then we switch to node $s^S_{i-1,j'}$, and follow the shortest path from there, i.e. we repeat task i on a different component. To be able to do so, we assume that at the end of each task, a checkpoint is written to stable storage (as is usual in backward recovery-based fault-tolerance [4]), which is kept until the follow-up task is completed successfully. Please note that $j' = k$ is possible, which explains why we considered nodes $s^S_{i,j}$ with $j \in S$ in the previous subsection.

IV. Experimental Results

We have implemented the two algorithms described in Section III:

- SP (Shortest Path) algorithm working on the execution graph for the fault-free case (Section III-A);
- FESP-1 (Single-Fault Extended Shortest Path) algorithm constructing and optimizing on the extended execution graph (Section III-B).

As baseline for comparison with FESP-1 in the single-fault scenario, we also implemented a simple brute-force enumeration algorithm that we call FSPX-1:

- FSPX-1 (SP for all eXclusions of single variants) considers for each non-failure-proof variant ($k = 1, 2, \ldots$) the reduced graph G_k derived from the execution graph by excluding just that one (assumed faulty) variant per task, performs SP on each G_k, and returns the set of resulting shortest paths.
  To achieve a fair comparison, we did some optimizations for this implementation, most notably to construct all sets of shortest paths together from one graph representation. In doing so, we became a little unfair towards our proposed method, e.g. we did not provide an access structure to find the shortest path depending on the fault situation, and thus did also not store the results in the brute-force implementation.

While the absolute runtimes, and thus the absolute saving in scheduling time, are small in our experiments, real-world graphs may be notably larger, e.g. have more phases, or more variants if multiple operating frequencies are used for each component.

Table I shows the wall-clock times (measured with Linux `gettimeofday`) for finding the shortest paths for 70 different combinations of the number of tasks ($n = 3, ..., 19$) and the number of variants per task ($p = 2, ..., 8$), averaged over 10 different task graphs with randomly generated edge weights (which model task variant runtimes and switching costs where applicable) for each such combination (n, p)[1], hence 700 task graph instances in total. The table also shows the number of nodes and edges of the execution graph, as well as the maximum and averaged actual number of edges in the extended graph used for FEXP-1. Column SP shows the

[1]We consider here that heterogeneous systems usually have not more than $p = 8$ different resource types in practice, and often fewer. The cutoff for the number n of tasks at 19 was an ad-hoc choice.

TABLE I
OPTIMIZATION TIMES IN MICROSECONDS FOR 70 DIFFERENT GRAPH SIZE CONFIGURATIONS WITH $n = 3 \ldots 19$ TASKS AND $2 \ldots 8$ VARIANTS PER TASK, AVERAGED OVER 10 GRAPHS (STANDARD DEVIATIONS IN BRACKETS) WITH RANDOMLY GENERATED EDGE WEIGHTS PER CONFIGURATION (n,p).

#Tasks	#Variants	Execution Graph		Extended Graph		Optimization Times			
(n)	(p)	#Nodes	#Edges	Max. #Edges	Avg. #Edges	SP	FSPX-1	FESP-1*	FESP-1
3	2	4	4	6	4.5	0.5 us [0.2]	0.6 us [0.2]	2.9 us [0.2]	0.5 us [0.2]
3	3	5	6	14	7.4	0.9 us [0.1]	1.4 us [0.2]	5.2 us [0.3]	1.1 us [0.1]
3	4	6	8	26	10.7	0.7 us [0.2]	2.3 us [0.3]	8.3 us [0.9]	2.2 us [0.3]
3	5	7	10	42	14.0	1.0 us [0.2]	3.4 us [0.3]	10.4 us [1.4]	1.8 us [0.3]
3	6	8	12	62	17.0	1.0 us [0.2]	3.8 us [0.5]	14.8 us [1.6]	4.7 us [1.6]
3	7	9	14	86	20.0	1.3 us [0.2]	5.1 us [0.6]	13.4 us [1.0]	3.6 us [0.3]
3	8	10	16	114	23.0	1.1 us [0.1]	5.2 us [0.1]	15.6 us [0.4]	4.8 us [0.2]
4	2	6	8	12	10.0	1.0 us [0.2]	1.1 us [0.1]	4.2 us [0.5]	0.6 us [0.2]
4	3	8	15	35	22.8	1.3 us [0.2]	2.0 us [0.0]	7.4 us [0.4]	1.7 us [0.2]
4	4	10	24	78	38.7	1.5 us [0.2]	4.4 us [0.2]	12.6 us [0.7]	3.7 us [0.4]
4	5	12	35	147	59.0	2.2 us [0.1]	7.3 us [0.2]	16.5 us [0.4]	4.4 us [0.3]
4	6	14	48	248	87.0	2.6 us [0.2]	10.9 us [0.2]	23.9 us [0.9]	6.5 us [0.6]
4	7	16	63	387	115.8	3.7 us [0.2]	16.2 us [0.3]	30.7 us [0.6]	9.5 us [0.5]
4	8	18	80	570	148.6	4.1 us [0.1]	22.4 us [0.3]	40.3 us [1.1]	12.7 us [0.9]
5	2	8	12	18	16.2	1.0 us [0.0]	1.0 us [0.0]	5.3 us [0.3]	1.2 us [0.1]
5	3	11	24	56	37.4	1.8 us [0.1]	2.9 us [0.1]	10.6 us [0.4]	2.7 us [0.3]
5	4	14	40	130	74.2	2.8 us [0.1]	6.4 us [0.2]	17.7 us [0.6]	5.0 us [0.4]
5	5	17	60	252	121.2	3.7 us [0.2]	11.3 us [0.2]	30.2 us [1.1]	10.0 us [0.8]
5	6	20	84	434	177.0	4.8 us [0.3]	19.1 us [0.5]	53.0 us [4.0]	16.2 us [0.6]
5	7	23	112	688	245.2	7.9 us [0.9]	36.6 us [3.8]	83.8 us [3.7]	23.4 us [1.9]
5	8	26	144	1026	321.1	9.2 us [1.0]	51.5 us [5.7]	84.7 us [4.1]	28.3 us [2.2]
7	2	12	20	30	26.6	1.8 us [0.1]	1.1 us [0.1]	7.0 us [0.4]	1.4 us [0.2]
7	3	17	42	98	77.0	3.4 us [0.3]	4.9 us [0.5]	20.0 us [1.7]	5.9 us [0.6]
7	4	22	72	234	160.2	6.0 us [0.7]	14.0 us [1.4]	47.0 us [3.7]	14.8 us [1.3]
7	5	27	110	462	259.2	6.7 us [0.6]	21.1 us [1.7]	63.2 us [6.1]	19.3 us [1.6]
7	6	32	156	806	410.5	9.2 us [0.7]	36.7 us [2.9]	93.5 us [6.4]	30.1 us [2.3]
7	7	37	210	1290	591.6	10.5 us [0.3]	50.2 us [1.0]	133.3 us [8.3]	45.2 us [2.2]
7	8	42	272	1938	789.3	17.8 us [1.8]	86.5 us [8.5]	168.4 us [11.0]	55.4 us [3.3]
9	2	16	28	42	39.0	2.2 us [0.3]	1.4 us [0.2]	9.3 us [0.9]	2.0 us [0.2]
9	3	23	60	140	113.4	3.7 us [0.2]	6.6 us [1.3]	22.8 us [0.7]	7.1 us [0.6]
9	4	30	104	338	258.2	6.1 us [0.2]	14.2 us [0.1]	50.9 us [1.9]	18.3 us [1.9]
9	5	37	160	672	437.6	9.3 us [0.2]	28.7 us [0.4]	95.0 us [4.9]	35.4 us [4.5]
9	6	44	228	1178	685.0	12.0 us [0.2]	46.8 us [0.7]	127.4 us [3.2]	44.2 us [1.1]
9	7	51	308	1892	1044.8	15.3 us [0.2]	73.4 us [0.6]	193.8 us [3.4]	66.0 us [1.1]
9	8	58	400	2850	1365.3	24.5 us [2.6]	140.5 us [17.5]	298.7 us [17.8]	98.0 us [4.5]
11	2	20	36	54	51.5	2.7 us [0.2]	1.7 us [0.2]	11.7 us [0.9]	3.2 us [0.4]
11	3	29	78	182	162.4	5.5 us [0.5]	7.4 us [0.6]	31.9 us [2.6]	9.2 us [0.6]
11	4	38	136	442	355.6	8.9 us [0.7]	19.2 us [1.5]	72.6 us [4.0]	24.3 us [0.9]
11	5	47	210	882	623.6	11.6 us [0.4]	37.3 us [1.2]	120.9 us [9.1]	39.5 us [2.5]
11	6	56	300	1550	1047.5	15.8 us [0.3]	63.2 us [0.9]	213.6 us [9.7]	77.3 us [6.8]
11	7	65	406	2494	1502.2	21.9 us [1.9]	106.6 us [9.6]	317.9 us [22.9]	110.2 us [8.7]
11	8	74	528	3762	2000.8	26.8 us [2.3]	153.8 us [12.6]	382.6 us [16.6]	136.1 us [9.0]
13	2	24	44	66	63.1	3.0 us [0.0]	1.8 us [0.1]	11.2 us [0.3]	2.9 us [0.2]
13	3	35	96	224	197.2	5.8 us [0.2]	8.3 us [0.2]	34.2 us [0.8]	10.7 us [0.5]
13	4	46	168	546	459.0	9.6 us [0.2]	22.0 us [0.4]	81.0 us [1.7]	27.9 us [0.7]
13	5	57	260	1092	861.6	18.6 us [3.1]	58.4 us [9.2]	173.9 us [10.2]	59.4 us [2.9]
13	6	68	372	1922	1385.0	19.3 us [0.3]	75.5 us [1.2]	260.8 us [12.3]	87.5 us [4.0]
13	7	79	504	3096	1989.6	26.4 us [2.2]	130.9 us [10.2]	375.4 us [12.0]	129.3 us [6.9]
13	8	90	656	4674	2872.2	41.2 us [4.3]	242.3 us [25.2]	636.0 us [54.4]	213.0 us [22.5]
15	2	28	52	78	76.0	3.9 us [0.4]	2.3 us [0.2]	14.6 us [1.2]	3.5 us [0.3]
15	3	41	114	266	243.0	6.5 us [0.2]	9.2 us [0.2]	44.8 us [1.0]	14.8 us [0.4]
15	4	54	200	650	566.0	11.2 us [0.2]	25.3 us [0.5]	104.7 us [5.8]	40.3 us [5.4]
15	5	67	310	1302	1047.6	16.3 us [0.3]	51.2 us [0.5]	195.1 us [11.5]	68.3 us [4.8]
15	6	80	444	2294	1691.5	25.2 us [2.3]	100.3 us [9.7]	326.9 us [21.9]	114.1 us [7.5]
15	7	93	602	3698	2440.4	38.2 us [3.8]	190.7 us [20.3]	556.0 us [40.0]	184.5 us [15.4]
15	8	106	784	5586	3726.8	56.5 us [4.7]	330.7 us [26.9]	840.9 us [73.3]	282.7 us [28.6]
17	2	32	60	90	87.7	4.7 us [0.5]	3.0 us [0.4]	17.7 us [1.2]	4.2 us [0.5]
17	3	47	132	308	281.2	10.9 us [1.6]	12.9 us [1.3]	65.1 us [5.0]	23.4 us [4.9]
17	4	62	232	754	675.4	15.6 us [1.9]	33.2 us [3.3]	140.3 us [8.7]	49.3 us [4.3]
17	5	77	360	1512	1243.6	28.2 us [2.6]	88.2 us [8.5]	264.6 us [23.3]	86.1 us [6.1]
17	6	92	516	2666	2051.0	34.3 us [3.3]	139.7 us [12.7]	487.5 us [18.9]	154.8 us [11.3]
17	7	107	700	4300	3113.8	47.0 us [4.8]	231.9 us [23.4]	715.8 us [57.8]	229.7 us [20.3]
17	8	122	912	6498	4374.9	46.4 us [3.7]	276.6 us [20.5]	779.3 us [21.8]	269.9 us [18.3]
19	2	36	68	102	99.3	4.1 us [0.1]	2.6 us [0.2]	17.5 us [1.3]	4.8 us [0.3]
19	3	53	150	350	326.0	10.0 us [1.1]	13.7 us [1.2]	66.5 us [3.8]	23.2 us [2.1]
19	4	70	264	858	756.9	20.6 us [1.8]	48.1 us [4.0]	173.0 us [8.7]	61.7 us [4.2]
19	5	87	410	1722	1436.0	36.4 us [2.3]	116.7 us [6.2]	364.9 us [23.8]	116.9 us [6.7]
19	6	104	588	3038	2441.0	38.2 us [3.7]	157.3 us [16.2]	572.2 us [30.7]	178.6 us [11.4]
19	7	121	798	4902	3615.0	53.3 us [5.1]	265.2 us [26.4]	778.8 us [68.7]	252.6 us [20.1]
19	8	138	1040	7410	5015.3	74.2 us [7.1]	438.3 us [39.9]	1178.4 us [89.1]	390.9 us [43.2]

averaged times for the SP algorithm computing the shortest path for the fault-free case (as in Figure 1). Column FSPX-1 gives the averaged times for computing shortest paths excluding one variant per node; these runs reuse the already allocated execution graph data structure and these times do not include the time for storing the $p-1$ different shortest path solutions. Finally, column FESP-1* gives the averaged times of the FESP-1 algorithm for constructing the extended graph *and* computing the shortest path for all possible failures of one execution unit (variant) among variants 1, 2, ..., 8, while column FESP-1 shows the time of the FESP-1 algorithm excluding the time for constructing the extended graph.

We observe that the optimization times of all three implementations grow with the corresponding graph size. The baseline algorithm FSPX-1 takes, except for cases with only 2 variants, longer time than SP, which is expected because it must execute SP multiple times on restricted execution graphs, despite our optimizations. Likewise, shortest path computation in FESP-1 takes longer than in SP, because it works on a much larger graph. Nevertheless, the average actual number of edges in the extended graph is clearly lower than the maximum possible number of edges, typically by a factor of up to 2, which shows the graph folding effect of the FESP-1 algorithm. FESP-1* takes generally longer than FSPX-1, which is mainly due to the construction of the much larger extended graph as we can see from the FESP-1 column while FSPX-1 reuses the execution graph data structure over all its iterations. It is actually fair to compare FSPX-1 and FESP-1 times, i.e. excluding graph construction times for both algorithms, because for FSPX-1 the graph construction time for the restricted graphs (and storing their solutions for its $p-1$ different fault scenarios) are not included in the FSPX-1 timings either. In addition, the measurement setup is too friendly towards FSPX-1 as the data structure is reused across all its $p-1$ SP runs and the solutions are not stored but overwritten. Based on that, we see that our FESP-1 algorithm considerably improves in performance in almost all cases over the FSPX-1 baseline, in the considered graph configurations with speedups typically between 10% and 100% for sufficiently large p.

V. Conclusions

We have presented how to extend a static scheduling algorithm for executing a multi-variant program on a heterogeneous computing platform with different execution units (like multiple CPU cores, a GPU, and other accelerators) to cover the situation of a fault of one or more components, so that one or more code variants are not executable anymore. Our scheduler finds, even for fault situations, the sequence of variants leading to minimum runtime, energy consumption, or a weighted sum of both. We have implemented and evaluated our scheduler and compared it to an alternative with synthetically generated program structures. In the experiments, our scheduler on average needs only 55% of the scheduling time of the alternative for 8 variants.

Future work will comprise the extension of the schedule's optimization target to constraint path scheduling, e.g. minimum energy under the constraint that the runtime stays below a pre-defined threshold, which is an NP-complete problem for general graphs [14]. Also, our implementation so far assumes that each variant runs on a different component, so that the fault of a component only affects one variant. Our hypothesis is that the advantage of our method increases with the number of tolerated faults, as there will be more opportunities to identify nodes of fault situations with other nodes. Finally, the graph size may grow exponentially with the number of unavailable variants. For example, for $n = 100$ stages and $p = 10$ variants, the graph comprises 1,000 nodes in the fault-free case, up to 10,000 if one variant may fail, up to 46,000 if two variants may fail, and already up to 130,000 if three variants may fail. We plan to quantify the limits of use for our approach in the worst case and for realistic task graphs taken from real-world software flows.

Beyond the concrete use in fault tolerance, the approach might also be used as a low-overhead anomaly detection in real-time scenarios.

Acknowledgment

We are very grateful to Martin Hellwig who mentioned a related optimization problem in route planning for electric vehicles, and thus sparked the present research.

C. Kessler acknowledges partial funding from EU H2020 project EXA2PRO (801015), exa2pro.eu.

References

[1] Q. Wu, Y. Ha, A. Kumar, S. Luo, A. Li, and S. Mohamed, "A heterogeneous platform with GPU and FPGA for power efficient high performance computing," in *Proc. 2014 International Symposium on Integrated Circuits (ISIC)*, 2014, pp. 220–223.

[2] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, pp. 258–266, 2005.

[3] L. Sha, "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, July/August 2001.

[4] D. P. Siewiorel and R. S. Swarz, *Reliable Computer Systems — Design and Evaluation*, 3rd ed. Natick, Mass.: A K Peters, 1998.

[5] S. Holmbacka and J. Keller, "Workload type-aware scheduling on big.LITTLE platforms," in *Proc. Internat. Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2017)*, S. Ibrahim, K.-K. R. Choo, Z. Yan, and W. Pedrycz, Eds. Cham: Springer International Publishing, 2017, pp. 3–17.

[6] R. Schlichting and F. Schneider, "Fail-stop processors an approach to designing fault-tolerant computing systems," *ACM Transactions on Computing Systems*, vol. 1, no. 3, pp. 222–238, 1983.

[7] P. Eitschberger, "Energy-efficient and fault-tolerant scheduling for many-cores and grids," PhD Dissertation, FernUniversität in Hagen, Germany, 2017.

[8] E. Hansson and C. Kessler, "Optimized variant-selection code generation for loops on heterogeneous multicore systems," in *Proc. ParCo-2015 conference, Edinburgh, UK, 1-4 Sep. 2015. Published in: G. Joubert, H. Leather, M. Parsons, F. Peters, M. Sawyer (eds.): Advances in Parallel Computing, Volume 27: Parallel Computing: On the Road to Exascale*. IOS Press, Apr. 2016, pp. 103–112.

[9] M. Korch and T. Rauber, "Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining," *J. Parallel Distrib. Comput.*, vol. 66, no. 3, pp. 444–468, Mar. 2006.

[10] M. Kicherer and W. Karl, "Automatic task mapping and heterogeneity-aware fault tolerance: The benefits for runtime optimization and application development," *Journal of Systems Architecture - Embedded Systems Design*, vol. 61, no. 10, pp. 628–638, 2015. [Online]. Available: https://doi.org/10.1016/j.sysarc.2015.10.001

[11] N. Msadek, R. Kiefhaber, and T. Ungerer, “A trustworthy, fault-tolerant and scalable self-configuration algorithm for organic computing systems,” *Journal of Systems Architecture*, vol. 61, no. 10, pp. 511 – 519, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S138376211500082X

[12] V. Izosimov, “Scheduling and optimization of fault-tolerant distributed embedded systems,” Doctoral dissertation, Linköping University, 2009.

[13] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[14] M. Ziegelmann, “Constrained shortest paths and related problems,” PhD Dissertation, Saarland University, Germany, 2001.