Energy-Efficient and Fault-tolerant Taskgraph Scheduling for Manycores and Grids

Patrick Eitschberger and Jörg Keller

FernUniversität in Hagen,
Faculty of Mathematics and Computer Science,
58084 Hagen, Germany
patrick.eitschberger@fernuni-hagen.de
joerg.keller@fernuni-hagen.de
www.fernuni-hagen.de/pv/

Abstract. Taskgraphs model a broad range of parallel applications. Despite static scheduling, processor failures can be overcome with the help of task duplication, which we explored in a previous proposal. With the advent of processor frequency scaling, energy can be saved by the runtime system as it is informed about gaps in the schedule and task dependencies, and thus can slow down processors as long as dependencies do not lead to a makespan increase. In the case of a fault, a makespan increase can be traded for additional energy investment by accelerating the task duplicates that run tasks from the crashed core. We evaluate our proposal with a large benchmark suite of taskgraphs with different sizes for a generic manycore architecture.

Keywords: taskgraph scheduling, task duplication, fault tolerance, frequency scaling, energy efficiency, power-aware computing, manycore computing, grid computing

1 Introduction

Many parallel applications can be decomposed into a set of tasks prior to execution. Thus they can be modeled by static taskgraphs, where each node is a task with a given runtime, and arcs (u, v) represent dependabilities, where a task u produces output of a known size upon completion, which must be transferred to task v that can only start if that input is available. For a given machine (we assume p identical processing units), a taskgraph must be scheduled, i.e. each task must be assigned to a processor with a given start time, when it is executed without interruption until it completes. For a valid schedule, at most one task may be active at any time, and the start times must represent the dependabilities between tasks, taking into account the communcation time if dependent tasks are mapped to different processing units. Typically, the goal of the scheduler is to minimize the makespan, i.e. the time when the last processing unit completes processing its assigned tasks.

At any time, a processing unit may fail, either because it is shutdown by its owner (a frequent situation in grids with contributed resources) or because of a

software fault (application or system hangs, a not too infrequent case in manycores with new architecture and thus new operating systems and compilers), or because of a hardware or network fault. In that case, redundancy must be available to continue processing of the taskgraph. As an alternative to temporal redundancy in the form of dynamic re-scheduling, which often leads to notably increased makespan, use of structural redundancy in the form of task duplication has emerged as a static technique to cope with failures. While it is clear that placements of tasks and duplicates on different processing units will ensure tolerance of one processor failure, the question of overhead in the fault-free case arises. While we developed techniques to avoid overhead by placing duplicates appropriately [4, 3], the additional processing of duplicates still increases the energy consumption of the taskgraph computation, which emerges as a new measure besides — and sometimes more important as — the makespan.

On the other hand, processor frequency scaling opens the dual possibility to save energy by slowing down tasks, thus filling gaps in the schedule without increasing makespan, and accelerating duplicates in case of a failure, to trade makespan increase against additional energy investment in this case. While frequency scaling typically is handled by the operating system according to usage rate, this is not appropriate for taskgraph computations as the local operating system has no information about the dependencies between tasks on processors belonging to different operating system instances. This information is available in the runtime system for the taskgraph computation, and manycores like Intel SCC delegate frequency scaling to user level and thus enable a reduction of energy consumption for our application scenario. Thus, the placement of tasks enables trading energy savings in the fault-free case against overhead in the fault case, and trading overhead in the fault case against additional energy investment. In contrast to other works, that optimize makespan for a given energy budget, our problem is to optimize energy consumption given a target makespan and the user preferences above.

Energy-efficiency in taskgraph scheduling has been considered in previous works. Kianzad et al. [7] consider frequency scaling in taskgraph scheduling, however they focus on integration of the scheduling and scaling steps. In contrast, we start with an existing schedule, insert duplicates for fault-tolerance and perform frequency scaling. Cong et al. [2] consider energy savings by exploiting input-dependent variations of task runtimes, i.e. they focus on dynamic frequency scaling, while we compute frequencies statically. Pruhs et al. [8] consider the situation where a certain energy budget is available for the taskgraph computation, and statically compute schedules with optimal makespan for that budget by scaling processor frequencies accordingly. Yet, none of those works considers fault-tolerance. In contrast, works that use task duplication for fault-tolerance, such as [5], do not consider energy consumption. While Unsal et al. [9] investigate energy aspects of fault-tolerance in real-time systems, they use application-level fault-tolerance techniques instead of task duplication. Thus, the combination of the three aspects is, to our knowledge, not known from the literature.

We compare several placement strategies for duplicates with the help of a simulator for a taskgraph runtime system [3] that we adapt to take frequency scaling into account and enable a forecast of energy consumption. The simulator assumes a generic manycore architecture with a standard model for dynamic power consumption, i.e. at frequency f, a core consumes power proportional to f^α, where $2 \leq \alpha \leq 3$, cf. e.g. [8]. We use $\alpha = 3$ based on previous experiments on the Intel SCC [1], but leave out constant factors and low-order terms for simplicity. As the frequency and thus the power consumption is fixed during execution of a task, the energy spent for that task is the product of power and task runtime.

As inputs, we take the taskgraphs from a benchmark suite of synthetic taskgraphs [6] that is organized according to several structural criteria, and thus allows to categorize results for certain types of taskgraphs. Our results indicate that the average energy improvement is between 30% and 80%, where 20% to 65% results from scaling down the frequency for idle times and up to 20% results from slow down tasks and thus scaling down the frequency for tasks.

The remainder of this paper is structured as follows. In Section 2 we present the extended scheduling algorithm. Section 3 presents and analyzes the simulation results. In Section 4, we conclude and give an outlook on open problems and future work.

2 Efficient Fault-Tolerant Scheduling

We start by reviewing the ideas from [4], and briefly introduce two extensions [3]. We concentrate on improving the energy efficiency of taskgraph schedules by scaling the frequencies of cores down, where tasks can be prolonged without any increase of the makespan. Furthermore we also improve the performance of the schedules in case of a fault, where the execution of the duplicates leads to an overhead and thus to a higher makespan. In these cases the duplicates can be accelerated by scaling up the frequency of the corresponding cores to reduce or completely undo the overhead.

2.1 Previous Approach

The approach of Fechner et al. [4] provides fault-tolerance to taskgraph schedules by task duplication. It starts with an already existing schedule (and taskgraph) and extends the schedule by including a duplicate task for each original task. The scheduling is done prior to execution, thus it is done statically. A duplicate (D) has to be placed on another Processing Unit (PU) than its corresponding original task, so that in case of a PU failure the schedule execution can be continued. We assume a fail-stop model, where the failure might be induced by a hardware, software, or network fault.

If the original task has finished its work it sends a commit message to the corresponding D, so that the D can abort its execution. Thus in every case either only the original task or only the D is finished and has to send its results to the

successor tasks. Because of the communication overhead, a D starts with a little delay called *slack*.

Mostly there are gaps in the schedule because of dependencies between the tasks. Especially those gaps can be used for an efficient placement of the Ds. In some cases the gaps might be too small to place a D. So the placement could lead to a shift of all successor tasks and thus to an overhead in the fault-free case. To avoid such an overhead and also to minimize the overhead in case of a fault, three different strategies for the placement are presented in [4].

In the first strategy only so called dummy duplicates (DD) are used. A DD is a placeholder for a D. It is placed with runtime 0 and only in case of a fault it is extended to a full duplicate task. Such a DD can be placed in gaps or between succeeding tasks. In this way, overhead in the fault free case can be avoided in general, but this strategy does not take much advantage of the gaps. In Fig. 1b an example placement of the DDs is illustrated. The corresponding taskgraph and input schedule are shown in Fig. 1a. For a better understanding the slack and the communication costs are disregarded.

The second strategy uses the schedules from the first strategy and converts DDs to Ds if there is a gap before the DDs. Only those DDs are converted that are placed at the end time of the corresponding original tasks. The size of the converted Ds is bounded by the length of the gap before the Ds. Thus in case of a fault a fraction of the D is already executed so that it only has to be extended for the remaining part of the original task. The modified schedule is illustrated in Fig. 1c. The second strategy uses the gaps more efficiently in comparison with the first strategy.

In the third strategy some of the gaps in the input schedule are partly extended to fill that gaps more efficiently with Ds. In this strategy a small overhead is allowed in the fault free case for a better overhead in case of a fault. Fig. 1d illustrates the modified schedule.

In [3] we firstly include the consideration of the communication times between the tasks, because those communication times influence the placement of the Ds and DDs. For that we assume to have a homogeneous network. The different communication times in the taskgraphs represent the different amounts of information that has to be transferred to the corresponding successor tasks. For example one task only needs an integer value, another task needs an array of floats, etc. A second extension in [3] is that we also consider task slowdowns as a special kind of fault. When the performance of a task decreases (because the PU is used for other duties, e.g. in grids with contributed resources), the corresponding D or DD is used instead if it can be finished earlier.

2.2 Energy efficiency improvements

In our previous work our goal was to guarantee no overhead in a fault free case and to have only a minimal overhead in case of a fault. But today and also in the future energy consumption might be more important to minimize. Usually the operating system offers some energy functions and automatically scales the processor frequency with respect to the usage rate. Especially for taskgraph

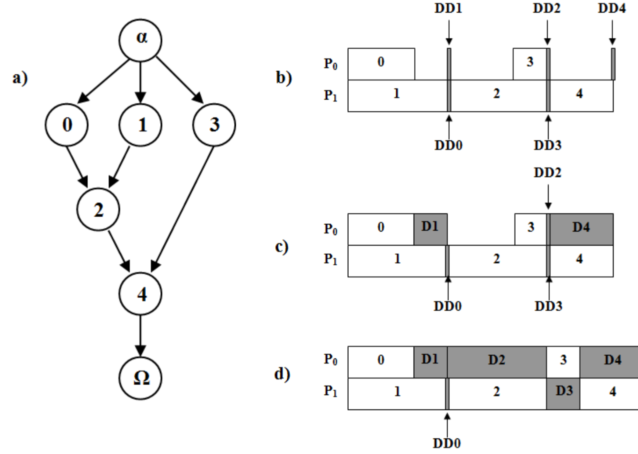

Fig. 1. An example taskgraph and three strategies in comparison.

scheduling such automatic energy functions cannot be used because the operating system has no information about the schedule and task dependencies and thus cannot decide if it can slowdown a task and scale down the core frequency without increasing the makespan of the whole schedule. In such a case only the scheduler resp. the underlying runtime system has the information about the taskgraphs and schedules. Thus the energy management has to be done there.

In our approach we implemented a simulator for a taskgraph runtime system that supports energy management functions. One example platform where one can use such a runtime system is the Single-Chip Cloud Computer (SCC), which is a research processor from Intel. The SCC consists of 24 dual-core tiles, i.e. 48 cores. With this platform, the user can scale the frequency and voltage of cores during the runtime of a program. The frequency scaling can be done for each tile, the voltage scaling only for groups of 8 cores. To calculate for each task the lowest possible frequency one has to check if a task could be extended or slowed down without increasing the makespan. It is important to note that the prolongation of a task could lead to a shift of the start time for some other tasks. But as long as these shifts do not have any influence on the makespan the extension does not have to be reduced. In the fault free case (where it is only senseful to extend or slowdown the tasks) only original tasks and their corresponding duplicates have to be considered. Because DDs are only placeholders for a duplicate, they are placed with runtime 0, and thus do not have to be extended. Duplicates on the other hand could also be slowed down or only be shifted, if the corresponding original task is slowed down too. But in case of only a shift the energy consumption would be higher than if the duplicate is also be slowed down. Thus, a slowdown of the duplicate in this case would be preferable to get the highest energy improvement. In the following the implementation of

the simulator is described before we also present techniques that use frequency scaling to improve the makespan and thus also the overhead in case of a fault.

2.3 Implementation

Our implementation generates and evaluates three schedules for each taskgraph: a basic schedule without duplicates and without frequency scaling, a schedule extended by duplicates, and a schedule where frequency scaling is applied to tasks and duplicates. The latter two are evaluated for the fault-free case (see below), and for the case of a fault (see next subsection).

As plenty of optimal and heuristic taskgraph schedulers are available, we did not integrate the scheduler, but input a taskgraph (in standard taskgraph format) and a corresponding schedule (in the corresponding schedule format). For this schedule, each task is assigned frequency $f_{normal} = 1$. This is a relative frequency, as all frequencies used in the sequel, to be multiplied with the frequency necessary to achieve the desired makespan (in wall-clock time). As we use the frequency only to compute energy consumption and relate different energy consumptions by percentages, using a relative frequency simplifies matters without any disadvantage. In each gap (between executing two tasks or after finishing the last task till the makespan), we assume a core to run at a frequency f_{idle}. For schedules without frequency scaling, we both consider $f_{idle} = f_{normal}$ and $f_{idle} \ll f_{normal}$.

The second schedule is generated by placing duplicates as explained in the previous subsection. The duplicates also are assigned frequency $f_{normal} = 1$.

A third schedule is generated by scaling all tasks and their duplicates, i.e. by assigning task frequencies f_a possibly smaller than f_{normal}, that still allow to finish the schedule by the makespan. As frequency scaling is a computational expensive non-linear optimization problem, we apply a greedy heuristic, to be explained below.

The energy consumption for each schedule is then computed for the fault-free case. For each task (and duplicate) T_a, its runtime t_a and frequency f_a are used to compute the task energy consumption $t_a \cdot f_a^3$ as the product of time and power consumption. For each gap, the energy consumption is computed similarly by using the length of the gap and the idle frequency f_{idle}. The total energy consumption is obtained by summing the energy consumption over all tasks, duplicates, and gaps.

The frequency scaling heuristic basically tries to scale down the frequency of a task followed by a gap of length x to $f_a < f_{normal}$ such that the task runtime t_a is increased by a so-called *buffer* $x' \leq x$, i.e. by a factor $f_{normal}/f_a = (t_a + x')/t_a$ such that the gap is filled as far as possible but timing constraints from dependencies are not violated. Consider first an example without duplicates depicted in Fig. 2. Task T1 with runtime 4 is followed by a gap of length 2, yet can only be extended by 1 time unit ($f_1 = 4/(4 + 1) = 0.8$ because of the communication to task T3 that needs 3 time units and must be completed by the start time 8 of task T3. Task T2 cannot be extended although it is followed by a gap, because the communication time to task T4 leaves no room. Task T3

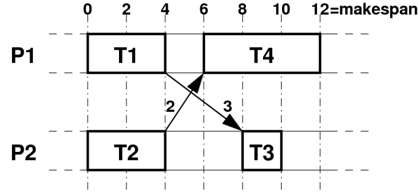

Fig. 2. Example schedule to illustrate the frequency scaling heuristic

can be scaled down to $f_3 = 0.5$ because the gap to the makespan has length 2. Task T4 cannot be scaled down as there is no gap afterwards. If the resulting task frequency is lower than f_{idle}, then it is set to f_{idle}. The tasks are considered one by one in the order given in the schedule.

A duplicate is treated like a task for computing the buffer. To avoid the situation that a task finishes later than its duplicate after frequency scaling, each task and its duplicate are scaled down to the same frequency, by using the minimum of the two buffers. Furthermore, if a task is followed by a gap and a duplicate, then the task can take over the duplicate task's buffer if it can use it without violating other constraints. This means that the duplicate must also be followed by a gap and its start time is increased accordingly, i.e. the gap is moved before the duplicate. Note that a dummy duplicate is treated like a duplicate with runtime 0.

2.4 Performance improvements in case of a fault

Until now we have only considered energy efficiency in the fault-free case. However, in case of a fault, avoiding a reduction of performance might be more important than the energy efficiency. We can use frequency scaling as well to improve the performance in such a case. When a PU crashes, it cannot be restarted as we use the fail-stop-model as explained above. Thus all following tasks on that PU will not be executed and the corresponding duplicates and dummy duplicates on the other PUs have to be extended and used to complete the schedule. Because of the extensions there might be an overhead that could result in a longer makespan. To minimize or totally undo this overhead one might also use some energy management functions.

If a D or DD has to be extended, the runtime system can save the current frequency (of that PU) and calculate the frequency that is needed to undo the overhead. This frequency is higher than f_{normal}, and thus leads to increased energy consumption. If the calculated frequency is higher than the supported maximum frequency of that PU, the frequency is set to the maximum frequency. Thus, the ability to undo the overhead largely depends on the frequencies that are supported by the PUs. After the execution of the corresponding extended D or DD the frequency can be scaled down to the previous frequency if the next task is an original task or a D that does not have to be extended. If the next task is a D or DD that has to be extended, the frequency can be scaled directly to the

new calculated frequency. If there is a DD that does not have to be extended and that is placed at the finish time of the prior task without a gap, the next task is considered. If the next task starts at the same time where the DD is placed (in this case there is no gap between the tasks) and if the next task has to be extended, the frequency can directly be scaled to that calculated frequency. Only in case of a gap the frequency could be scaled down to the previous frequency. To avoid some of the high frequencies in case of a fault one could use the buffers from the previous subsection.

3 Experimental Results

We evaluate the energy-efficiency of our proposal with a benchmark suite of synthetic taskgraphs [6] comprising 72,000 optimal schedules that differ in the number of PUs (2, 4, 8, 16 and 32), the number of tasks (7 - 12, 13 - 18 and 19 - 24), edge density, the edge length and the node and edge weights. Furthermore we also used a simple list scheduler, that maps the tasks on that PU where they can start their execution first. For 36,000 taskgraphs, the number of tasks is as above, to see how energy consumption for non-optimal schedules with larger gaps can be improved. For 36,000 tasks graphs, the number of tasks is up to 250 to see how our proposal works for larger schedules.

There are totally three different variants for each strategy that result from the consideration of the communicaton times in our previous work [3]. Variant a) is the placement like explained above. In variant b) we use so called waitdummies to reduce the overhead in a fault free case and in variant c) we consider some of the communication times only in a fault case so that we can already quarantee no overhead in a fault free case. In Fig. 3 the improvements for the different strategies and variants and also for the original schedules without any fault-tolerant aspects (strategy 0) are presented for the different test benchmarks (TB). In these experiments we used the value 0.1 for f_{idle}.

Each bar in Fig. 3 is divided into two parts. The dark part represents the improvements of scaling down the frequency to $f_{idle} = 0.1$ in gaps, the light part represents the improvements of using the buffers to slowdown the tasks by scaling down their frequencies. In total the average energy improvements for the different testsets vary from around 30% for the testset TB-Optimal up to around 75% for the testset with the large testcases generated with the simple list scheduler. As expected most of the improvements result from the idle times (between 20% and 65%). The results reflect also that the schedules of the "TB-Optimal" testset are highly optimized and have only a few resp. short gaps in comparison to the other testsets. With using also the buffers, the energy improvements can be increased up to additional 20%. The highest improvements are for strategy 1 in all variants (abc). This results from only using DDs in the first strategy that do not need additional space and usually can be shifted without increasing the makespan. For the strategies 2 and 3 (abc) the improvements are lower, because in these cases there are a lot of Ds and thus fewer or smaller gaps.

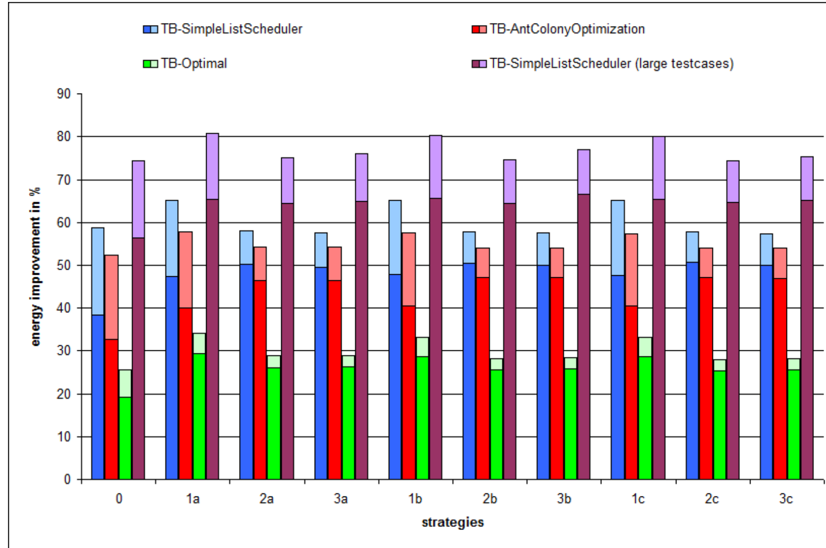

Fig. 3. Average Energy improvement

In Fig. 4 the overhead improvements in case of a fault are presented for the test benchmarks with small schedules, averaged over all schedules and all possible fault positions, cf. [3]. As settings we used $f_{idle} = 0.1$ and 2.0 as maximum possible frequency. The overhead can be reduced by 27% to 38%. The highest improvements show in strategy 1 because the DDs have not been slowed down in the fault free case and thus most of the DDs can still be extended without increasing the makespan. In strategies 2 and 3 the overhead improvement is smaller for a better energy efficiency in the fault free case, but the improvement is still high as only the duplicates are speeded up that have to be extended in case of a fault. We could also see that using only part of the buffer for energy reduction in the fault-free case leads to a reduced overhead in the fault case, however we cannot detail this for lack of space.

4 Conclusions and future work

We have presented a static task scheduling algorithm that uses task dupliction to handle processor failures and also increases the energy efficiency in a fault free case by scaling the core frequency down for some tasks without increasing the makespan. On the other hand we also used the frequency scaling to improve the performance of the schedules in case of a fault by scaling up the frequency for duplicates that have to be extended. Our results indicate that frequency scaling is worthwhile for both, improving the energy efficiency for schedules with a given makespan and also for the performance of the underlying runtime system in case of a fault. As future work, we plan to explore optimization possibilities

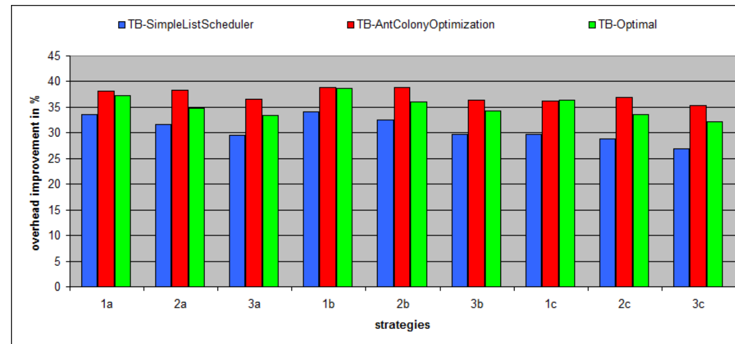

Fig. 4. Overhead improvement in the fault case.

by deviating from the order scheduling–duplicate placement–scaling, either by combination of steps or different orders. Finally, we plan to extend our experiments from simulations to a prototype system that we can use on the Intel SCC or Kalray MPPA, which allow user-level frequency scaling with the restriction that only groups of cores can scale their frequencies.

References

1. Cichowski, P., Keller, J., Kessler, C.: Modelling Power Consumption of the Intel SCC. In: Proc. 6th MARC Symposium, pp. 46–51, 2012.
2. Cong, J., Gururay, K.: Energy Efficient Multiprocessor Task Scheduling under Input-dependent Variation. In: Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE '09), pp. 411–416, 2009.
3. Eitschberger, P., Keller, J.: Effcient and Fault-Tolerant Static Scheduling for Grids. In: Proc. 14th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2013), 2013
4. Fechner, B., Hönig, U., Keller, J., Schiffmann, W.: Fault-Tolerant Static Scheduling for Grids. In: Proc. 13th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS'08), 2008
5. Hashimoto, K., Tsuchiya, T., Kikuno, T.: Effective Scheduling of Duplicated Tasks for Fault Tolerance in Multiprocessor Systems. IEICE Transaction on Information and Systems, E85-D(3):525–534, 2002.
6. Hönig, U., Schiffmann, W.: A comprehensive test bench for the evaluation of scheduling heuristics. In: Proc. 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS04), pp. 437-442, 2004
7. Kianzad, V., Bhattacharyya, S., Ou, G.: CASPER: An Integrated Energy-Driven Approach for Task Graph Scheduling on Distributed Embedded Systems. In: Proc. 16th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'05), July 2005.
8. Pruhs, K., van Stee, R., Uthaisombut, P.: Speed Scaling of Tasks with Precedence Constraints. Theory of Computing Systems, 43(1):67–80, July 2008.
9. Unsal, O., Koren, I., Krishna, C.M.: Towards Energy-Aware Software-Based Fault Tolerance in Real-Time Systems. In: Proc. International Symposium on Low Power Electronics and Design (ISLPED '02), pp. 124–129, 2002.