A Practical Approach for Energy Efficient Scheduling in Multicore Environments by Combining Evolutionary and YDS Algorithms with Faster Energy Estimation

Zorana Banković, Umer Liqat, Pedro López-García

A Practical Approach for Energy Efficient Scheduling in Multicore Environments by combining Evolutionary and YDS Algorithms with Faster Energy Estimation

Zorana Banković[1], Umer Liqat[1] and Pedro López-García[1,2]

[1] IMDEA Software Institute, Madrid, Spain
[2] Spanish Council for Scientific Research (CSIC), Spain
{zorana.bankovic,umer.liqat,pedro.lopez}@imdea.org

Abstract. Energy efficient scheduling and allocation in multicore environments is a well-known NP-hard problem. Nevertheless approximated solutions can be efficiently found by heuristic algorithms, such as evolutionary algorithms (EAs). However, these algorithms have some drawbacks that hinder their applicability: typically they are very slow, and if the space of the feasible solutions is too restricted, they often fail to provide a viable solution. In this paper we propose an approach that overcomes these issues. The approach is based on a custom EA that is fed with predicted information provided by an existing static analysis about the energy consumed by tasks. This solves the time inefficiency problem. In addition, when this algorithm fails to produce a feasible solution, we resort to a modification of the well-known YDS algorithm that we have performed, well adapted to the multicore environment and to the situations when the static power becomes the predominant part. This way, we propose a combined approach that produces an energy efficient scheduling in reasonable time, and always finds a viable solution. The approach has been tested on multicore XMOS chips, but it can easily be adapted to other multicore environments as well. In the tested scenarios the modified YDS can improve the original one up to 20%, while our EA can save $55-90\%$ more energy on average than the modified YDS.

Keywords: Scheduling, energy efficiency, multicore systems, evolutionary algorithms, YDS.

1 Introduction

Energy efficient scheduling has gained a lot of interest in the recent past. A great number of publications, e.g., [6], try to present it as a mixed integer linear optimisation problem, which can be solved using mixed integer linear programming, or using a heuristic approach. However, these algorithms become impractical or fail to deliver a solution as the problem size grows. There is a significant group of

publications on using EAs for the problem of optimal scheduling and allocation in multiprocessor systems that allow Dynamic Voltage and Frequency Scaling (DVFS), e.g., the approach presented in [11] aims to minimize both energy and makespan as a bi-objective problem.

Energy efficient scheduling and allocation in multicore environments with enabled DVFS is a well-known NP-hard problem. Nevertheless approximated solutions can be efficiently found by heuristic algorithms, such as evolutionary algorithms (EAs). An example is our previous EA-based algorithm [3]. In our setting, we want to solve the general scheduling problem where the tasks have arbitrary release times and deadlines, and where preemption and migration of tasks are allowed, but the problem still remains NP-hard for arbitrary release times and deadlines of the tasks which are not *agreeable*,[1] as it was proven in [1]. Our algorithm has been adapted for its application to multicore XMOS chips, but it could easily be adapted to any multicore environment, ranging from small scale embedded systems up to large scale systems, such as data centers. Its first practical implementation relied on an existing analytical model for calculating the energy consumption for programs running on these chips [8]. The energy model is limited to the cases when all the cores belong to the same chip, thus they must run at the same voltage and frequency level at each moment. For this reason, in this work we solve the problem of the so-called *global* DVFS, when all the cores always have the same voltage and frequency.

In our first EA implementation, each individual in each generation is evaluated by using the above mentioned program energy model, which requires the execution traces of the programs. Given that the traces can be huge, even for small programs, such evaluation introduces a huge amount of overhead. In order to overcome this issue, in this paper we use an existing static analysis which, at compile time, without the need of executing the programs, and in a few seconds, gives a safe estimation of the energy consumed by programs. As energy consumption often depends on (the size of) input data, which is not known at compile time, the static analysis provides the energy as a function of the input parameters, which is calculated once such input values are known at runtime. The energy consumption estimated by using the static analysis for a given scheduling is computed as the sum of the energies of the tasks running on different cores. This gives a safe upper bound on the total energy consumption, although it may be less precise than the estimations computed with the program energy model mentioned previously. This may reduce possible energy savings, nevertheless, the information it provides is still good enough to decide which scheduling is better, and the gain in speed of the algorithm is huge: the simulation time is reduced from a few hours to a few minutes.

However, the EAs can have trouble in finding a viable solution, in the sense that not all task deadlines are met, if the task deadlines are too tight. In order to overcome this problem, we have adapted the standard YDS [2] algorithm [15] (explained in Section 2) to multicore environments. As we will see later, our

[1] Two tasks are *agreeable* if the task with later release time also has a later deadline.
[2] The name is created using the first letter of the authors' last names.

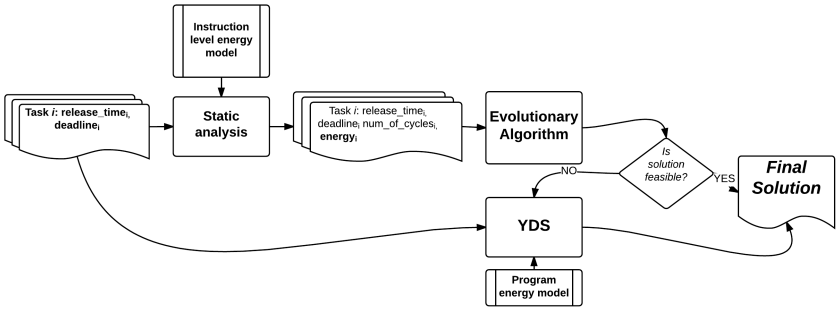

Fig. 1. Overview of our scheduling approach.

experimental results show that if the EA finds a viable solution, it is better than the one obtained by our modified YDS algorithm in terms of energy savings.

For these reasons, the approach we propose (depicted in Figure 1) consists of the following steps:

1. Perform the static analysis of the input tasks to estimate the energy consumed by each of them.
2. Execute the EA using such estimations.
   - If the EA provides a viable solution, i.e., all the task deadlines are met, this is the final solution.
   - Otherwise, execute our modified YDS algorithm and take its output as the final solution.

We can distinguish two energy models in Figure 1:

- *Instruction-level energy model:* gives an estimation of the energy consumed by the execution of a single instruction, which in general depends on its inputs, context, etc. However, for simplicity, the model assigns a constant value to each instruction. It is used by an abstract-interpretation based static analysis to infer the energy consumed by a program.
- *Program-level energy model:* a formula that gives the energy consumption of the entire program, as presented in [8], which is used by our modified YDS algorithm.

In summary, in this paper we propose a time efficient scheduling approach, which always provides a viable energy efficient solution, and scales well as the input size grows. The main original contributions of our approach are:

1. The combination of an EA algorithm that resorts to a modified YDS algorithm, which always provides a viable solution.
2. Use of static analysis for energy estimation at compile time to guide the EA process, which results in significant speed-up in solving the scheduling problem, and hence the practicability of our approach, while still providing a solution with significant energy savings.

3. An improvement of the YDS algorithm, which efficiently solves the static power issue in the situations the chip cannot be switched off (explained in Section 2).

The rest of the paper is organised as follows. Section 2 gives more details about our proposed approach. Section 3 presents an experimental evaluation of the approach. Finally, some conclusions are drawn in Section 4.

2 Our Proposed Approach

Evolutionary Algorithm The approach we propose is based on the NSGA-II [4] multiobjective evolutionary algorithm with two objectives: the execution time and the total energy consumption, where both should be minimised. The objectives are clearly in conflict, since the application of DVFS reduces energy, but increases execution time. This justifies the usage of a multiobjective algorithm. NSGA-II has been proven in the literature to perform good when the number of objectives is small [3], and in our case we have only two.

Since the output of the multiobjective approach is a set of solutions which form the (approximated) Pareto front, we can choose the solution that meets some given energy and/or time requirements. Usually we pick the solution with the minimal energy consumption among those that meet the given time bound (if applicable).

Individual Representation The problem that we are solving is the optimal (in terms of energy or time, possibly under some requirements involving them) allocation and scheduling of a set of tasks, where each task is defined by its:

- Unique *ID*.
- *Release time*, i.e., the moment when the task becomes available.
- *Deadline*, i.e., the latest moment when the task has to finish.
- *Number of clock cycles*, as a good approximation of the execution time.

Thus, the solution to this problem has to contain the following information:

- The core(s) where each task will be executed. Since we allow task migration, a task can be allocated to more than one core.
- The current voltage and clock frequency (V, f) state to exploit DVFS.
- The time periods when the tasks are executed.
- The number of clock cycles each task will execute in the different periods marked by the preemption and migration of that task. This allows to express the number of cycles a task will execute before it is preempted, as well as the number of cycles it will execute after it is resumed in the same or in a different core, etc.

Having in mind these requirements, we have designed the solution representation as shown in Figure 2, which does not introduce significant overhead when executing the EA. Any given task has a positive (unique) number as its ID.

Each gene representing a task ID is followed by a gene representing the number of cycles of the task that will be executed without any preemption. The order of task IDs represents the order of their temporal execution. We also use negative two digit numbers to encode the spatial allocation of the tasks. The first digit represents the core where the tasks are being executed and the second one an encoding of the (V, f) state of that core. As it will be explained in Section 3, Table 1, the number of different cores and states is finite, as well as the number of their combinations. The tasks following the allocation code are executed on that coded location. For instance, on Figure 2 we read: on core 1 in state 2, 48 cycles of task 1 will be executed, and 77 cycles of task 5, in this exact order, etc.

Our approach allows a random order of allocation codes, in order to solve the most general problem. However, if two consecutive allocation codes have different (V, f) states, this means that the tasks allocated to the cores they represent will not be executed in parallel, since all of the cores have to run at the same (V, f) at any moment, and thus the (V, f) state has to be changed before the second group of tasks is executed. For example, in Figure 2 the allocation code following -12 is -24, which means that the chip will be first in the (V, f) state 2 and all tasks allocated to core 1 will be executed sequentially on that core. After they finish their execution, the core will change its (V, f) state from 2 to 4, and the tasks allocated to core 2 will be executed sequentially on core 2. If the second allocation code were -22 instead of -24, then the (V, f) state would not change, and the tasks allocated on cores 1 and 2 would be executed in parallel.

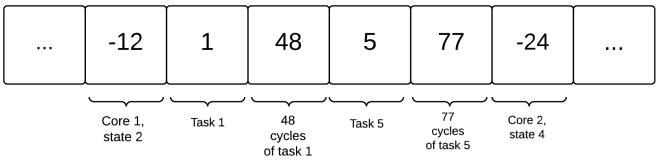

Fig. 2. An example of (part of) a solution (i.e., individual) representation.

Population Initialisation Individuals in the initial population are created by randomly assigning tasks to random cores in random (V, f) settings with equal probability. However, in order to provide a load balanced solution (as much as possible), the probability of choosing a core decreases as its load increases, which is given by the following formula:

$$Prob = \frac{1}{NumberOfCores} - \frac{CurrentCoreLoad}{TotalLoad} \tag{1}$$

where $CurrentCoreLoad$ stands for the current load of the core expressed as the number of cycles, while $TotalLoad$ stands for the total number of cycles of all the tasks on all cores. According to the formula, at the beginning of the initialisation process, all cores have the same probability of being chosen, while this probability decreases as the core becomes loaded, and is close to 0 when the

load reaches the state where it is equally distributed in all cores. If during the initialisation process a newly calculated probability value of a core is below 0, the value is rounded to 0, and no new load will be assigned to that core. These random solutions do not always have to provide a viable solution, i.e., some of the tasks might miss their deadlines. For this reason, the mutation operator and the objectives are designed to deal with this problem.

Solution Perturbations Given the unique nature of the individual representation, we have designed new crossover and mutation operators. The individuals that participate in the crossover are selected by using the standard tournament selection process.

The Crossover Operator Since our solution allows task migration, a given task ID can appear more than once, so we cannot apply any of the existing permutation-based crossover operators. Thus, we have designed our own operator, where each child will preserves the order of genes representing the task allocation and scheduling from one parent, and only the genes representing the number of cycles can be taken from the other parent. In this way, the produced offspring is a combination of both parents, and is at the same time a viable solution to the problem. The process is depicted in Figure 3 for the most simple case of 2 cores, 2 tasks and 2 states, with one possible output. We can observe that the first child, $C1$ takes the scheduling and allocation from the first parent, $P1$, and the cycle distribution from the second parent, $P2$, while the second, $C2$, takes the scheduling and allocation from $P2$ and the cycle distribution from $P1$.

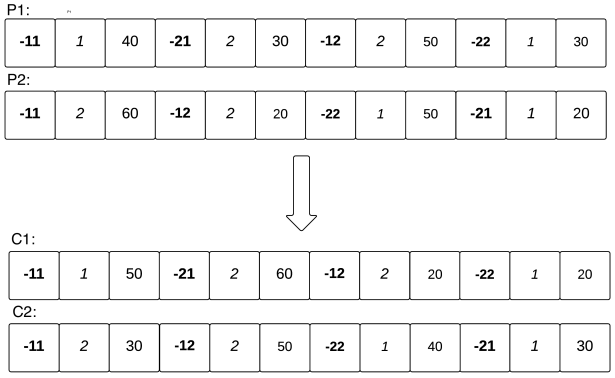

Fig. 3. An example of a crossover operation.

The Mutation Operator The mutation operator can perform different actions involving one or two tasks. Consider two tasks i and t. When choosing the first one we give higher probability to the tasks which miss their deadlines, in order to achieve a viable solution as soon as possible. In each generation we perform

either one of the following operations with the same probability (depicted in Fig. 4):

- *Swapping:* i and t, together with their corresponding number of cycles, exchange their positions in the solution. However, in order to avoid creating solutions which are not viable, i and t have to belong to the cores that are executed in parallel (defined with consecutive allocation codes that are in the same (V, f) state). In Fig. 4 we can observe that tasks 1 and 2 are swapped between cores 1 and 2, and both cores are in state 1.
- *Moving*: move i to a random position j. For the same reason as before, the position j has to belong to a core being executed in parallel as i's. In Fig. 4 we can observe that the first part of task 1 (40 cycles) is moved to core 2, before task 2.
- *Changing the number of cycles:* assigns a different number of cycles to all the appearances of task i, in a way the total number of cycles of that task remains the same. In Fig. 4 we can observe that task 1 after the change executes 25 cycles on core 1 in state 1 and 45 cycles on core 2 in state 2.

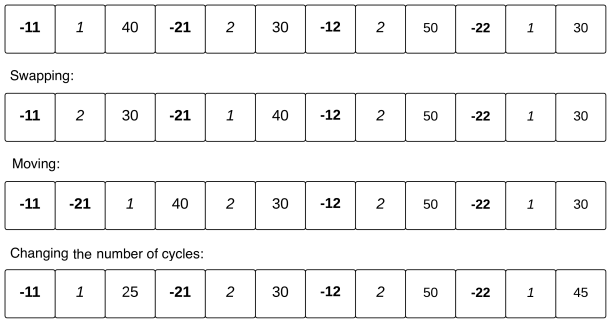

Fig. 4. Examples of mutation operations.

Objective Functions

Execution Time One objective of our optimisation problem is to minimize the total execution time of the schedule, which is the time spent since the first task starts its execution until the last task finishes its execution. However, since the initial population is randomly created, it is possible that some of the tasks miss their deadlines, making the solution unviable. Assuming that these solutions can provide some quality genetic material, we do not want to discard them completely, but we penalize them by adding the amount of time the tasks have missed their deadlines to the objective function. Thus, the time objective function for n cores and k different tasks is the following:

$$\hat{T} = T + \sum_{1 \leq i \leq n} \Big(\sum_{1 \leq j \leq k} x_{i,j} \cdot y_j \cdot (s_{i,j} + \tau_{i,j} - deadline_j) \Big) \tag{2}$$

where T is the total execution time, given by:

$$T = \max_{\substack{1 \le i \le n \\ 1 \le j \le k}} (x_{i,j} \cdot (s_{i,j} + \tau_{i,j})) - \min_{\substack{1 \le i \le n \\ 1 \le j \le k}} (x_{i,j} \cdot s_{i,j}) \tag{3}$$

where $s_{i,j} \geq 0$ is the moment when task j is scheduled on core i ($s_{i,j} = 0$ if the task j is not scheduled on core i), $\tau_{i,j}$ is the execution time of task j on core i, $x_{i,j}$ is a binary value, that represents whether the task j is executed on core i ($x_{i,j} = 1$) or not ($x_{i,j} = 0$). The second part of formula (2) represents the penalisation, where y_j is another binary value that expresses whether the task j has missed its deadline, $deadline_j$, ($y_j = 1$) or not ($y_j = 0$).

Energy Consumption This objective represents the total energy consumption of the given schedule. In the most general case it is given by the following formula:

$$E = \sum_{1 \le i \le n} (P_{st,i} \cdot T + \sum_{1 \le j \le k} (x_{i,j} \cdot p_{i,j} \cdot \tau_{i,j})) \tag{4}$$

where $P_{st,i}$ is the static power of core i, T is the same as in formula (3), $p_{i,j}$ is the dynamic power of task j when executed on core i, and $x_{i,j}$ and $\tau_{i,j}$ are the same as in formula (2). In this work we use static analysis to estimate the energy of single tasks, which will be explained in more detail in Section 2, while the energy is the sum of the energies of all the tasks, as given in formula (4).

The Modified YDS Algorithm YDS [15] is a well known algorithm for energy-efficient scheduling for single core DVFS-enabled environments. We have chosen it because it always finds a feasible (and optimal) solution that minimises the total energy consumption, it is simple and fast. However, it does not take into account the static power, which nowadays forms an important part of the total power. YDS reduces the frequency and voltage in order to minimise the dynamic power in a way that the execution time of tasks are extended to their deadlines. However, this also results in an increase of the static energy. Thus, there is a critical point from which further reduction of voltage and frequency actually starts increasing the energy consumption. Attempts to reducing the static energy when applying DVFS have mainly tried to group the inactive periods by moving task executions towards their deadline or its release point, and turning off the chip during such periods [7]. However, this is not always possible since chip wake up can take more time than available. Our alternative proposal does not turn off the chip, but instead, finds the critical (V, f) point below which further decrease is not beneficial. Thus, our modification of YDS consists of the following steps:

1. In order to decide if it is beneficial to further decrease the frequency, and in this way avoid the problem when the increase in energy consumed by the static power leads to the total energy increase, we use the simple slope-based method presented in [12].
2. In order to support the multicore system, we propose two different heuristics for allocating the tasks to different cores: the load balanced solution and the solution where a task allocation leads to the minimal frequency increase. After the allocation, the YDS algorithm is applied to each core.

3. In order to adapt the algorithm so that it assigns only the frequencies supported by the system, we propose to divide the computational load into two parts and execute them on two supported frequencies in a way the total execution time remains (almost) the same.

A Solution to the Static Power Issue of YDS As mentioned before, we use the simple slope-based method presented in [12]. The only requirement for its application is the availability of a power model where the static and dynamic power are separated. The main idea of the method is the following. If we fix the voltage, the power is a linear function of the frequency, and after applying some simple numeric transformations (explained in [12] in detail), it can be expressed in this way:

$$P_f = P_{f_{min}} + m \cdot (f - f_{min}) \tag{5}$$

where P_f denotes that the power depends on frequency f, and f_{min} is the minimal possible frequency, assumed to be the one which permits the execution of tasks to finish at their deadlines, and m is called the slope of the power function. If we can compare energies at different frequencies, we will know if it is energy efficient to decrease the frequency. Since power is a function of m, the same applies to energy, and thus it determines the decision of decreasing the frequency. In theory, there should exist a slope at which energy is equal for all frequencies. This slope is called the *critical power slope*:

$$m_{critical} = \frac{P_{f_{min}} - P_{idle}}{f_{min}} \tag{6}$$

If the actual slope m (calculated from Eq. 5) is greater than the critical one then we can decrease the frequency in order to save energy. However, if the slope is lower than the critical one, then the frequency should be increased in order to save the energy. Since the voltage can also change, the slope should be calculated for each (V_x, f_x) point for each frequency f_x:

$$m^{f_x}_{critical} = \frac{P_{f_x} - P_{idle}}{f_x} \tag{7}$$

This value is then compared with the actual slope m^{f_x} at each (V_x, f_x) point with frequency f_x. Again, if $m^{f_x} > m^{f_x}_{critical}$, we should decrease the frequency in order to save the energy. However, if $m^{f_x} < m^{f_x}_{critical}$, the frequency should be increased in order to save the energy.

Optimal Task-Core Allocation Other aspects of adapting YDS to a multicore environment consist of finding an optimal number of cores, and allocation, i.e., assignments of tasks to cores. In this work we do not deal with the first part of the problem, we just show that an optimal number of cores exists and it is not necessarily equal to the maximal possible number of cores. Regarding the second part, we have tested two possibilities:

1. Assign tasks to the cores so that the load is equally distributed between them.

2. Assign tasks in the way its addition assumes minimal change in frequency: a task t is assigned to the core with minimal activity, measured as the number of clock cycles in its active period $(t\_release\_time, t\_deadline)$.

Assigning Frequencies Supported by the System If the frequency f calculated by YDS is not supported by the system, the total number of cycles ω_i is divided in two parts, ω_{i1} and ω_{i2}, which are executed on two frequencies f_1 and f_2 ($f_1 \leq f \leq f_2$) supported by the underlying system. The values of ω_{i1} and ω_{i2} are calculated by solving the following system of equations:

$$\begin{aligned} \frac{\omega_i}{f} &\approx \frac{\omega_{i1}}{f_1} + \frac{\omega_{i2}}{f_2} \\ \omega_i &= \omega_{i1} + \omega_{i12} \end{aligned} \tag{8}$$

Energy Static Analysis as Input In order to estimate the energy consumed by programs without actually running them we use an existing static analysis. It is a specialisation of the generic resource analysis presented in [13] for programs written in a high-level C-based programming language, XC [14], running on the XMOS XS1-L architecture, that uses the instruction-level energy cost models described in [10]. The analysis is general enough to be applied to other programming languages and architectures (see [10, 9] for details). It enables a programmer to symbolically bound the energy consumption of a program P on input data $\bar{x}$ without actually running $P(\bar{x})$. It is based on setting up a system of recursive cost equations over a program P that capture its cost (energy consumption) as a function of the sizes of its input arguments $\bar{x}$. Consider for example the following program written in XC:

```
int fact(int N) {
  if (N <= 0) return 1;
  return N * fact(N - 1);
}
```

The transformation based analysis framework of [10, 9] would transform the assembly (or LLVM IR) representation of the program into an intermediate semantic program representation (HC IR), that the analysis operates on, which is a series of connected code blocks, represented as Horn Clauses. The analyser deals with this HC IR always in the same way, independent of where it originates from, setting up cost equations for all code blocks (predicates).

$$fact_e(N) = fact\_if_e(0 \leq N, N) + c_{entsp} + c_{stw} + c_{ldw} + c_{ldc} + c_{lss} + c_{bf}$$

$$fact\_if_e(B, N) = \begin{cases} fact_e(N-1) + c_{bu} + 2\ c_{ldw} + c_{sub} + \\ \qquad + c_{bl} + c_{mul} + c_{retsp} & \text{if } B \text{ is } \texttt{true} \\ c_{mkmsk} + c_{retsp} & \text{if } B \text{ is } \texttt{false} \end{cases}$$

The cost of the function $fact$ is captured by the equation $fact_e$ which in turn depends on the equation $fact\_if_e$, that captures the cost of the two clauses representing the two branches of the *if* statement, and a sequence of low-level

Table 1. Viable (V, f) pairs for XMOS chips.

$Voltage(V)$	0.95	0.87	0.8	0.8	0.75	0.7
$frequency(MHz)$	500	400	300	150	100	50

instructions. The cost of low-level instructions, which constitute an energy cost model, is represented by c_i where $i \in \{entsp, stw, ldw, ...\}$ is an assembly instruction. Such costs are supplied by means of assertions that associate basic cost functions with elementary operations.

If we assume (for simplicity of exposition) that each instruction has unitary cost in terms of energy consumption, i.e., $c_i = 1$ for all i, we obtain the energy consumed by `fact` as a function of its input data size (N): $fact_e(N) = 13\ N + 8$.

The functions inferred by the static analysis are arithmetic functions (polynomial, exponential, logarithmic, etc.) that depend on input data sizes (natural numbers). We use them in our scheduling and allocation algorithm to estimate the energy consumed by the different tasks involved. Such estimation can be computed very efficiently once the input data sizes of the tasks are known, since all the basic arithmetic functions involved can be evaluated in little bounded time.

3 Experimental Evaluation

XMOS Chips In this work we target the XS1-L architecture of the XMOS chips as a proof of concept. Although these chips are multicore and multithreaded, in this work we assume a single core architecture with 8 threads, which is the architecture for which we have an available energy model. In this case, we can use the algorithm and representation of individuals described in Section 2 by considering that a thread in our experiments is conceptually equivalent to a core executing tasks sequentially, as described previously. We refer the reader to [3] for a description of a representation of individuals whose allocation codes include three digits, representing: a core, a thread running in parallel on that core, and a (V, f) state.

In the XS1-L architecture, the threads enter a 4-stage pipeline, meaning that only one instruction from a different thread is executed at each pipeline stage. If the pipeline is not full, the empty stages are filled with $NOPs$ (no operation). Effectively, this means that we can assume that the threads are running in parallel, with frequency F/N, where F is the frequency of the chip, and $N = max(4, numberOfThreads)$. DVFS is implemented at the chip level, which means that all the cores have the same voltage and frequency at the same time. In order to apply DVFS, we need a list of Voltage-Frequency (V, f) pairs or ranges that provide a correct chip functioning. We have experimentally concluded that the XMOS chips can function properly with the voltage and frequency levels given in Table 1.

Task Set In order to test our proposed approach, we use two different groups of task sets. The first group is made up of small tasks, where the EA training with the program-level energy model takes around one day to complete. This group is used to show the difference between the results obtained with the EA trained with the program-level energy model and the EA trained with the energy estimations obtained by the static analysis. In this group, we use four different arithmetic programs: `fact(N)`, for calculating the factorial of N, `fibonacci(N)` for calculating the Nth Fibonacci number, `sqr(N)` for computing N^2 and `power_of_two(N)` for computing 2^N. In total, we have created a set of 22 tasks to be scheduled, corresponding to the execution of the previous programs with different inputs N.

In the second group we use real world programs, where the EA training based on the program-level energy model is not practical: `fir(N)`, i.e., Finite Impulse Response (FIR) filter, which in essence computes the inner-product of two vectors of dimension `N`, a vector of input samples, and a vector of coefficients, and `biquad(N)`, which is a part of an equaliser implementation, based on a cascade of Biquad filters, whose consumed energy depends on the number of banks `N`. We have used four different FIR implementations, with different number of coefficients: 85, 97, 109 and 121. Furthermore, we have used four implementations of the biquad benchmark, with different number of banks: 5, 7, 10 and 14. We have tested our approach in scenarios of 16 and 32 tasks, each one corresponding to such implementations. The tasks corresponding to the same implementation have different release times and deadlines.

The energy consumed by the programs is inferred at compile time by the static analysis described in Section 2. Such energy is expressed as a function of a parameter N, the size of the input, which is only known at runtime. Such functions are given in Table 2 for 3 of the 6 different voltage and frequency levels used in this work (for conciseness, as the functions for each program have the same complexity order, but different coefficients). The static analysis assumes that a single program (task) is running on one thread on the XMOS chip, while all other threads are inactive. In this implementation, the EA algorithm approximates the total energy of a schedule by adding the energies of all the tasks. Although in this way we loose precision, the estimation still provides precise enough information for the EA to decide which schedule is better.

Testing Scenarios In our current implementation, we assume no dependency between the tasks since it is not supported by the available energy models. The release times and deadlines of the different tasks are set in different scenarios in order to experimentally show the benefits of DVFS and optimal scheduling, where all the tasks have different release times and deadlines, with tighter deadlines; and that it is important to take into account the static power, especially in the case of loose deadlines.

Scenario 1: Tasks with Loose Deadlines In this scenario the *release time* of a task k, denoted T^k_{rel} is a random moment between 0 and the total execution time at the maximal frequency of all the tasks executed sequentially on a single core. Also, the *deadline* of a task is a random moment between $T^k_{rel} + 10 \times T^k_{maxf}$

Table 2. Energy functions inferred by static analysis for 3 different pairs of voltage (V)/frequency (MHz).

	V = 0.70 **F = 50**	**V = 0.75** **F = 100**	**V = 0.80** **F = 150**
$fact(N)$	$60.5\ N + 46$	$35\ N + 26.7$	$27\ N + 20.5$
$fib(N)$	$87.19 \times 1.62^N +$ $26.7 \times (-0.62)^N - 74.7$	$50.32 \times 1.62^N +$ $15.44 \times (-0.62)^N - 43$	$38.68 \times 1.62^N +$ $11.85 \times (-0.62)^N - 33.2$
$sqr(N)$	$21.3\ N^2 + 121\ N$ $+39.1$	$12.3\ N^2 + 69.8\ N$ $+22.5$	$9.48\ N^2 + 53.7\ N$ $+17.3$
$powerOf2(N)$	$55.1 \times 2^N - 39$	$63.7 \times 2^N - 39$	$24.49 \times 2^N - 30$
$fir(N)$	$74.93\ N + 124.5$	$43.36\ N + 71.9$	$33.41\ N + 55.2$
$biquad(N)$	$386\ N + 128$	$223.6\ N + 74.2$	$172.5\ N + 57.2$

and $T^k_{rel} + 20 \times T^k_{maxf}$, where T^k_{maxf} denotes the execution time of the task at maximum frequency. This way we achieve a scenario with loose deadlines even at a smaller frequency.

Scenario 2: Tasks with Tight Deadlines Here, the *release time* is the same as in Scenario 1. However, the *deadline* of a task is a random moment between $T^k_{rel} + 5 \times T^k_{maxf}$ and $T^k_{rel} + 7 \times T^k_{maxf}$. This way we get tighter deadlines, but also provide a set of tasks which are schedulable on the given platform. Note that the deadlines become even tighter as the frequency decreases.

Results: The Improved YDS Table 3 shows the savings of our improved YDS algorithm presented in Section 2 compared to the original YDS, for different number of cores and for two different ways of task allocation: Alloc. 1, where the load is evenly distributed between the cores, and Alloc. 2, where the addition of a task implies a minimal increase in the frequency. The energy saving resulting from a particular scheduling is calculated using the following formula:

$$\frac{YDS\_original - YDS\_modified}{YDS\_original} \cdot 100 \tag{9}$$

In Table 3, energy savings are achieved in all, but in two cases with tight deadlines. A possible reason could be the fact that all the threads need to have the same frequency always, which means that the maximal necessary frequency is assigned, which is not necessarily be optimal for all the threads. However, in the case of loose deadlines, the savings are much more significant, since the static power plays a more important role.

Results: EA vs. improved YDS Both EA and YDS are implemented in C++. EA extends the MOGAlib library [5] for multiobjective genetic algorithms. In the EA, the population of 200 individuals is evolved for 150 generations. The probability of both crossover and mutation is 0.9. The mutation is assigned higher probability than usual due to its important role for reaching a viable solution. Since the result of the optimisation process is a set of possible solutions

Table 3. Energy savings obtained by the modified YDS vs. the original YDS (%).

	Tight deadlines		**Loose deadlines**	
#Cores	*Alloc. 1*	*Alloc. 2*	*Alloc. 1*	*Alloc. 2*
1	4.18	4.18	6.21	6.21
2	1.5	4.26	14.67	14.67
3	-5.26	3.17	14.67	14.67
4	2.22	2.77	8.8	8.8
5	-3.28	3.47	11.18	11.18
6	0.95	4.34	11.82	11.82
7	4.8	3.03	10.9	10.9
8	19.36	5.61	10.56	10.56

which form the approximated Pareto front, we take the solution with minimal energy consumption where all task deadlines are met.

Table 4 presents results comparing the EA trained with the energy estimations provided by static analysis, versus the improved YDS algorithm presented in Section 2. In the first column, the energy of the final solution calculated by using the program-level energy model is given (EA_s). The second column gives the energy of the final scheduling obtained by the modified YDS algorithm (referred as YDS_m) using the program-level energy model, while the third column gives the energy saving of the EA trained with static analysis compared to YDS. Finally, the last column shows the energy saving obtained with the EA trained with the program-level energy model(EA_m) [2], which is only applicable in the scenarios with a small number of numeric tasks. Each row shows statistics for each scenario taken from 10-20 runs of the algorithm for the same scenario, where $CI0.01$ and $CI0.05$ represent 99% and 95% confidence intervals, meaning that we can claim with 99(95)% certainty that the final result will fall in these intervals.

In order to perform the comparison between the EA and YDS_m, in the case of EA and tight deadlines, we present the results when the EA can find a viable solution. However, the EA does not always provide a viable solution in all the scenarios with tight deadlines created as explained previously. As we can see in Table 4, if the EA finds a viable solution, it always performs better than the YDS_m. We can also observe that the EA trained with the program-level energy model achieves better results. However, the EA trained with the energy estimations by static analysis still achieves very good results, but with the training process that lasts around 10 minutes, compared to around 24 hours of training the EA with the program-level energy model, which makes it much more practical.

4 Conclusions and Future Work

In this work we propose a holistic approach for optimal scheduling, allocation and voltage(V) and frequency(f) assignment in multicore environments, adapted for

Table 4. EA vs. improved YDS in different scenarios.

	$EA_s(\mu J)$	$YDS_m(\mu J)$	$\frac{(YDS_m - EA_s)}{YDS_m}(\%)$	$\frac{(YDS_m - EA_m)}{YDS_m}(\%)$
A scenario with 22 small numeric tasks and loose deadlines				
Mean	14.3	33.1	56.8	76.57
CI 0.01	11.6 - 17	NA	48.64 - 64.95	67.87-85.27
CI 0.05	12.2 - 16.4	NA	50.45 - 63.14	70.05- 83.09
A scenario with 22 small numeric tasks and tight deadlines				
Mean	14.6	34.8	60.92	69.83
CI 0.01	11.5-17.7	NA	49.14 - 66.95	57.18-57.18
CI 0.05	12.2 - 17	NA	51.15 - 64.94	60.34-60.34
A scenario with 16 tasks made of Biquad and FIR filters and loose deadlines				
Mean	4.38	35.3	87.59	NA
CI 0.01	3.4 - 5.3	NA	85 - 90.37	NA
CI 0.05	3.7 - 5.1	NA	85.55 - 89.52	NA
A scenario with 16 tasks made of Biquad and FIR and tight deadlines				
Mean	14.5	35.4	59.04	NA
CI 0.01	9.4- 19.6	NA	44.63 - 73.45	NA
CI 0.05	10.6 - 18.4	NA	48.02 - 70.06	NA
A scenario with 32 tasks made of Biquad and FIR filters and loose deadlines				
Mean	17.85	68.16	73.81	NA
CI 0.01	10.8 - 25	NA	63.32 - 84.15	NA
CI 0.05	12.5 - 23.3	NA	65.82 - 81.66	NA
A scenario with 32 tasks made of Biquad and FIR filters and tight deadlines				
Mean	29.43	68.16	56.82	NA
CI 0.01	0.72 - 51.6	NA	24.3 - 89.44	NA
CI 0.05	12.5 - 46.3	NA	32.07 - 81.66	NA

multicore XMOS chips. The main part of our approach is based on our custom developed EA-algorithm, which relies on static analysis to efficiently estimate the energy of input tasks. The use of such static analysis improves significantly the speed of the EA training process, thus allowing its real world applicability. Furthermore, since in the case of very tight task deadlines the EA can fail in providing a feasible solution, we add one more stage based on the YDS algorithm, which has been adapted for multicore environments and improved in a way it takes into account the static power. In this way, we have developed an efficient approach which is capable of providing energy savings in each possible scenario.

However, although the use of static analysis based estimations still provides energy savings, we have seen that better results can be achieved with more precise energy estimations. For this reason, we plan to use a static analysis of the energy consumed by concurrent programs, which is expected to provide additional savings. As a future work, we also plan to study the effect of different versions of the crossover and mutation operators in different situations, which could enable adding a heuristic for choosing the optimal version for each possible scenario.

Acknowledgements. The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement 318337, ENTRA - Whole-Systems Energy Transparency, Spanish MINECO TIN'12-39391 *StrongSoft* and TIN'08-05624 *DOVES* projects, and the Madrid M141047003 *N-GREENS* project.

References

1. S. Albers, F. Müller, and S. Schmelzer. Speed scaling on parallel processors. In *Proc. of SPAA '07*, pages 289–298, USA, 2007. ACM.
2. Z. Banković and P. López-García. Energy Efficient Allocation and Scheduling for DVFS-enabled Multicore Environments using a Multiobjective Evolutionary Algorithm. In *Proc. of GECCO '15*. ACM, 2015. To Appear.
3. Z. Banković and P. Lopez-Garcia. Stochastic vs. Deterministic Evolutionary Algorithm-based Allocation and Scheduling for XMOS Chips. *Neurocomputing*, 150:82–89, 2015.
4. Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2000.
5. Balázs Gaál. *Multi-Level Genetic Algorithms and Expert System for Health Promotion.* PhD thesis, Univ. of Panonia, Faculty of Information Technology, 12 2009.
6. Marco E. T. Gerards et al. Analytic clock frequency selection for global DVFS. In *Proc. of PDP '14*, pages 512–519, 2014.
7. R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proc. of DAC '04*, pages 275–280. ACM, 2004.
8. S. Kerrison and K. Eder. Measuring and modelling the energy consumption of multithreaded, multi-core embedded software. *ICT Energy Letters*, pages 18–19, July 2014.
9. U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder. Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR. Technical report, ENTRA Project, April 2014. Appendix D3.2.4 of Deliverable D3.2. Available at `http://entraproject.eu`.
10. U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M.V. Hermenegildo, and K. Eder. Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Proceedings of LOPSTR'13*, 2014.
11. Mohand Mezmaz et al. A bi-objective hybrid genetic algorithm to minimize energy consumption and makespan for precedence-constrained applications using dynamic voltage scaling. In *Proc. IEEE CEC '10*, pages 1–8. IEEE, 2010.
12. Akihiko Miyoshi et al. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proc. of ICS '02*, pages 35–44, USA, 2002. ACM.
13. A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP, ICLP'14 Special Issue*, 14(4-5):739–754, 2014.
14. D. Watt. *Programming XC on XMOS Devices.* XMOS Limited, 2009.
15. F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, 0:374, 1995.