An evolutionary scheduling approach for trading-off accuracy vs. verifiable energy in multicore processors

U. LIQAT*, *IMDEA Software Institute and Universidad Politécnica de Madrid (UPM), Spain.*

Z. BANKOVIĆ**, *IMDEA Software Institute, Madrid, Spain.*

P. LOPEZ-GARCIA†, *Spanish Council for Scientific Research (CSIC) and IMDEA Software Institute, Madrid, Spain.*

M. V. HERMENEGILDO‡, *IMDEA Software Institute and Universidad Politécnica de Madrid (UPM), Spain.*

Abstract

This work addresses the problem of energy-efficient scheduling and allocation of tasks in multicore environments, where the tasks can allow a certain loss in accuracy in the output, while still providing proper functionality and meeting an energy budget. This margin for accuracy loss is exploited by using computing techniques that reduce the work load, and thus can also result in significant energy savings. To this end, we use the technique of loop perforation, that transforms loops to execute only a subset of their original iterations, and integrate this technique into our existing optimization tool for energy-efficient scheduling. To verify that a schedule meets an energy budget, both safe upper and lower bounds on the energy consumption of the tasks involved are needed. For this reason, we use a parametric approach to estimate safe (and tight) energy bounds that are practical for energy verification (and optimization applications). This approach consists in dividing a program into basic ('branchless') blocks, establishing the maximal (resp. minimal) energy consumption for each block using an evolutionary algorithm, and combining the obtained values according to the program control flow, by using static analysis to produce energy bound functions on input data sizes. The scheduling tool uses evolutionary algorithms coupled with the energy bound functions for estimating the energy consumption of different schedules. The experiments with our prototype implementation were performed on multicore XMOS chips, but our approach can be adapted to any multicore environment with minor changes. The experimental results show that our new scheduler enhanced with loop perforation improves on the previous one, achieving significant energy savings (31% on average for the test programs) for acceptable levels of accuracy loss.

1 Introduction

It is well-known that optimization of task scheduling and allocation (for energy efficiency) in multicore environments is an *NP*-hard problem. However, it can often be efficiently solved in practice with heuristic techniques, e.g., our approach based on Evolutionary Algorithms (EAs) [4].

*E-mail: umer.liqat@imdea.org
**E-mail: zorana.bankovic@imdea.org
†E-mail: pedro.lopez@imdea.org
‡E-mail: manuel.hermenegildo@imdea.org

In this work, we turn our attention to the problem of scheduling tasks optimally where such tasks allow introducing a certain degree of accuracy loss, and energy budgets have to be met. Numerous applications, such as video streaming or machine learning, can tolerate certain levels of accuracy loss, while still meeting the expected quality levels in the observable results. A technique that allows trading performance for accuracy is loop perforation [19], which in essence consists in skipping every n-th loop iteration, for a given n. A decrease in accuracy requirements then allows reductions in computational load, leading to both an increase in performance and a decrease in energy consumption. Thus, in this article we trade Quality of Service (QoS) for energy, since accuracy can be considered one aspect of QoS. More specifically, we tackle the following scheduling problem in multicore systems: given a set of tasks with known release times and number of cycles to compute them, find a proper allocation and scheduling of the tasks (allowing task migration), as well as a (V,f) assignment (i.e. a voltage and frequency pair) for the cores, in such a way that the total energy consumption is minimized (ensuring that a given energy budget is met), and accuracy is maximized (ensuring that a minimal acceptable accuracy level is also met).

In our approach, different levels of accuracy are achieved by applying the loop perforation technique with different values for the parameter n mentioned above. Hence, we deal with two competing objectives: accuracy and energy. Accuracy is defined in terms of deviation of the output after applying the loop perforation. To estimate the energy consumption corresponding to a given schedule, and to be able to verify/certify that an energy budget is met, we use a technique that infers safe upper and lower bounds on the energy consumed by each of the tasks involved.

Our previous work [3] used average energy models, which give an average energy consumption of each task in a schedule and hence an average energy consumption of the whole schedule. Herein we extend our energy-efficient scheduling algorithm by using upper- and lower-bound energy models that provide safe upper and lower bounds on the energy consumption of each task and hence on the whole schedule. Such bounds on task energy consumption are obtained by using our approach [12] that infers parametric upper and lower bounds on the energy consumption of a program by using a combination of static and dynamic techniques. The dynamic technique, based on an evolutionary algorithm, is used to determine the maximal/minimal energy consumption of each basic block. A static analysis is then used to combine the energy values obtained for the blocks according to the program control flow, and produce energy consumption bounds of the whole program. These bounds on energy consumption of a program are then used in our scheduling approach for energy-efficient scheduling to have a safer approximation of the energy consumption of a schedule.

As a proof of concept, in this article we focus on XMOS multicore chips, which provide support for Dynamic Voltage and Frequency Scaling (DVFS) at the chip level (i.e., all cores have the same voltage and frequency at the same time). However, we believe that our results can be easily generalized to other multicore environments.

The rest of the article is organized as follows. Section 2 put our work in context with respect to the related work. Then, Section 3 details our proposed approach for evolutionary scheduling. Section 4 presents an experimental evaluation of the approach and, finally, some conclusions are drawn in Section 5.

2 Related work

In the existing literature, techniques that include QoS as an objective in scheduling are mainly designed for Grid or Cloud Computing environments, where QoS is measured as either execution time, cost, etc., and has to be provided according to the signed Service Level Agreement (SLA) between the provider and the customer [21–23]. Multiobjective genetic algorithms were used in [23] to minimize cost and execution time, since they can be in conflict. A similar approach is presented in [22]. In the recent past, energy consumption has become a bottleneck, and techniques to reduce it have been developed, such as [21], where the authors try to minimize energy and maximize QoS at the same time in a Cloud Computing environment. The multiobjective optimization problem is solved using particle swarm optimization. However, as far as we know, none of the approaches in the literature propose to trade off QoS (accuracy in our case) with energy or performance in a scheduling problem by using a transformation of the program code, in our case, by using loop perforation.

There is a significant group of publications that use EAs for the problem of optimal scheduling and allocation in multiprocessor systems that allow DVFS. For example, the approach presented in [17] aims to minimize both energy and makespan as a bi-objective problem. The same problem is solved in other work [16], but using the island model of parallel GA populations. Another approach [11] addresses the problem from two opposite points of view: in the first one, it optimizes the energy given the scheduler length, while in the other one it optimizes the scheduling length given the energy bound. However, none of these solutions include the possibility of two levels of parallelism as we do in this article (which is an extension of our previous work [2]), where each processor can have a number of different threads executing in parallel. In addition, we also introduce the possibility of task migration.

3 The proposed evolutionary scheduling approach

In this section, we describe our proposed evolutionary scheduling approach for trading-off accuracy vs. energy in multicore environments. First, the loop perforation technique, that we use for reducing energy consumption down to the allowed accuracy level, is described in Section 3.1. Then, Section 3.2 describes our custom multiobjective evolutionary algorithm for scheduling. Finally, in Section 3.3 we outline our technique for estimating upper and lower bounds on the energy consumption of a schedule, that is based on the combination of static analysis of programs/tasks, with dynamic modeling of basic blocks.

3.1 Loop perforation

In general, the loop perforation technique transforms loops to execute a subset of their original iterations. In this article we use a particular case of this technique, *modulo perforation*, that skips every n-th iteration [19]. In this case, the perforation rate r, which represents the expected percentage of loop iterations to skip, is determined by $r = \frac{1}{n}$. As we will see in Section 3.2, in our representation of individuals we use the value n, that we call the *skipping iteration number*. Such number can be varied to trade off accuracy vs. energy, so that for higher values of n, fewer iterations are skipped, which implies that the work load, the energy consumption and the accuracy of the computation are higher. Conversely, more energy is saved for lower values of n, at the cost of growing accuracy loss. This trade-off between accuracy and energy consumption justifies the usage of a multiobjective algorithm. As we will see in Section 3.2, loop perforation is one of the possible actions of our mutation operator.

3.2 EA

In this article we extend our previous work [4], where we developed a custom multiobjective evolutionary algorithm (MOEA) for scheduling which is a dominance-based approach called

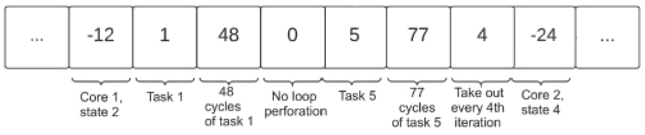


FIG. 1. Individual representation

Non-dominated Sorting Genetic Algorithm (NSGA-II) [8]. There exist also two other classes of approaches for MOEAs: aggregation-based (e.g. MOEA/D-M2M [15]) and indicator-based (e.g. HypE [1] and SMS-EMOA [5]). Some of these are known to perform better, particularly for many-objective optimization (four or more objectives). The NSGA-III algorithm [7] has also been proposed for many-objective optimization problems. In our case, we have two conflicting objectives, accuracy and energy consumption, since we want to decrease the energy consumption as much as possible while maintaining the accuracy level as high as possible (and always above a given threshold). The non-dominated solutions are generated using NSGA-II, while the EA follows the standard steps of evolutionary algorithms: initialization and evolution, where we implement a standard tournament selection, and customized crossover and mutation operators. In the following, we give more details about the particular improvements carried out in this work.

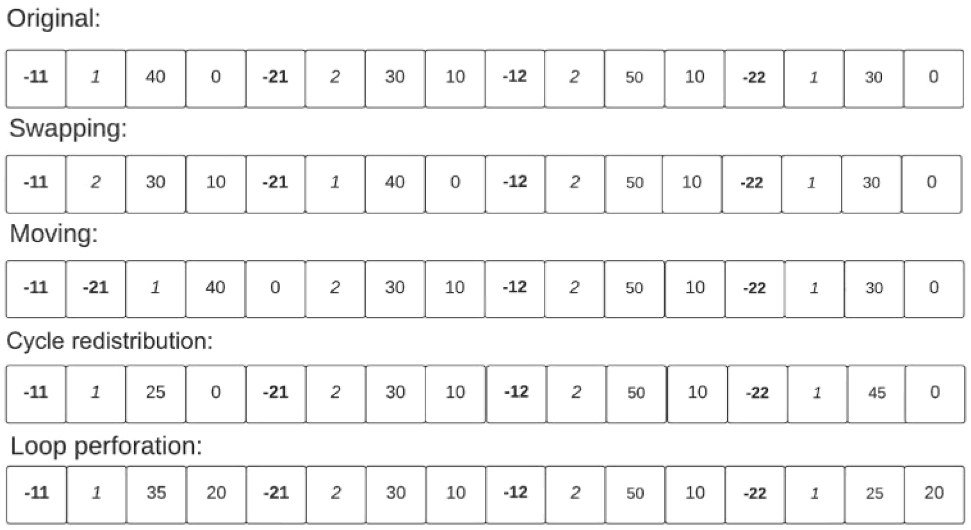
Individual. A solution to the problem we are solving, i.e., an individual, contains information about the scheduling and allocation of each task, how many cycles of each task are executed in the current run (to support task migration), and voltage and frequency levels of the cores at each moment (the allowed (V,f) pairs are represented as 'state' codes). Moreover, since in this work we add the possibility of decreasing accuracy by using loop perforation, we introduce another field: the *skipping iteration number* n, as explained in Section 3.1. For a given task, loop perforation is applied by using such *skipping iteration number* (and hence the corresponding perforation rate) on the loop(s) previously identified as tunable (i.e. filtering out critical loops whose perforation causes the computation to produce unacceptable results, crash, increase its execution time or execute with a memory error).

An example of a part of an individual is given in Figure 1, which can be read as follows: on core 1 in state 2, we execute in this order,

- 48 cycles of task 1, without applying loop perforation on it (its skipping iteration number is zero), and
- 77 cycles of task 5, with loop perforation skipping every 4th iteration.

Population initialization. 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. The number of cycles of a task executed in each run, as well as the skipping iteration number, are randomly chosen.

The crossover operator. In our customized crossover operator, the order of appearance of the tasks and their allocation are taken from only one of the parents, so that the child preserves such information. However, the child can take the distribution of the number of cycles and the skipping iteration number from any of the parents with equal probability.


Original:

-11	*1*	40	0	**-21**	*2*	30	10	**-12**	*2*	50	10	**-22**	*1*	30	0

Swapping:

-11	*2*	30	10	**-21**	*1*	40	0	**-12**	*2*	50	10	**-22**	*1*	30	0

Moving:

-11	**-21**	*1*	40	0	*2*	30	10	**-12**	*2*	50	10	**-22**	*1*	30	0

Cycle redistribution:

-11	*1*	25	0	**-21**	*2*	30	10	**-12**	*2*	50	10	**-22**	*1*	45	0

Loop perforation:

-11	*1*	35	20	**-21**	*2*	30	10	**-12**	*2*	50	10	**-22**	*1*	25	20

FIG. 2. Different possibilities for mutation

The mutation operator. In each generation, one of the following actions is performed with the same probability (see Figure 2 for an example):

- *Swapping:* tasks i and j, together with their corresponding number of cycles and skipping iteration number, change their positions in the solution. To avoid creating solutions which are not viable, tasks i and j must be assigned to cores that are in the same (V,f) state. In Figure 2, tasks 1 and 2 are swapped between cores 1 and 2 (both cores are in state 1).
- *Moving*: randomly move task i to another core and (V,f) state. For the same reason as before, the new (V,f) state must be the same as i's original state. In Figure 2, the first part of task 1 (40 cycles), originally allocated to core 1, is moved to core 2 (before task 2), and both cores are in state 1.
- *Cycle redistribution:* randomly change the distribution of the number of cycles of task i between its appearances on the different cores. In Figure 2, task 1 now executes 25 cycles on core 1 in state 1 and 45 cycles on core 2 in state 2.
- *Loop perforation:* randomly choose a task i, assign a skipping iteration number to it, decrease the total number of cycles accordingly to the cycles skipped by loop perforation, and (also randomly) distribute them among the existing appearances of task i in the solution. In Figure 2 no loop perforation was performed on task 1 in the original solution (i.e. its skipping iteration number was zero). However, the application of this mutation assigns a skipping iteration number of 20 to task 1. As a result of loop perforation, its total number of cycles in the original solution, 70, distributed between cores 1 (40) and 2 (30), is decremented to 60, with a different distribution between cores 1 (35) and 2 (25).

Objective functions: energy consumption. This objective represents the total energy consumption of the given schedule, and it should be minimized. It is given with the following formula:

$$E = \sum_{1\le i\le n} (P_{st,i}\cdot T + \sum_{1\le j\le k} (x_{i,j}\cdot p_{i,j}\cdot \tau_{i,j})), \tag{3.1}$$

where $P_{st,i}$ is the static power of core i, T is the total execution time of the schedule, i.e., the moment when the last task finishes its execution, $\tau_{i,j}$ is the execution time of task j on core i, $x_{i,j}$ is a binary value, $x_{i,j} \in \{0, 1\}$, that represents whether the task j is executed on core i ($x_{i,j} = 1$) or not ($x_{i,j} = 0$), and $p_{i,j}$ is the power of task j when executed on core i.

Objective functions: accuracy. We define the accuracy as an average deviation of the output after applying loop perforation, and it should be minimized. If a task performs some sort of signal processing, where the output is a digital signal consisting of a number of samples, the deviation is calculated as the Euclidean distance between the outputs obtained with and without loop perforation.

3.3 Inferring energy bounds statically by evolutionary analysis of basic blocks

To approximate upper and lower bounds on the energy consumption of a schedule, we use the approach that we proposed in [3]. It combines (dynamic) energy modelling techniques, to infer energy bounds on the program's basic blocks by using an EA, with static analysis techniques, that use these bounds on basic blocks to infer bounds on the whole program as a function of its input data sizes.

3.3.1 Energy modelling of basic blocks

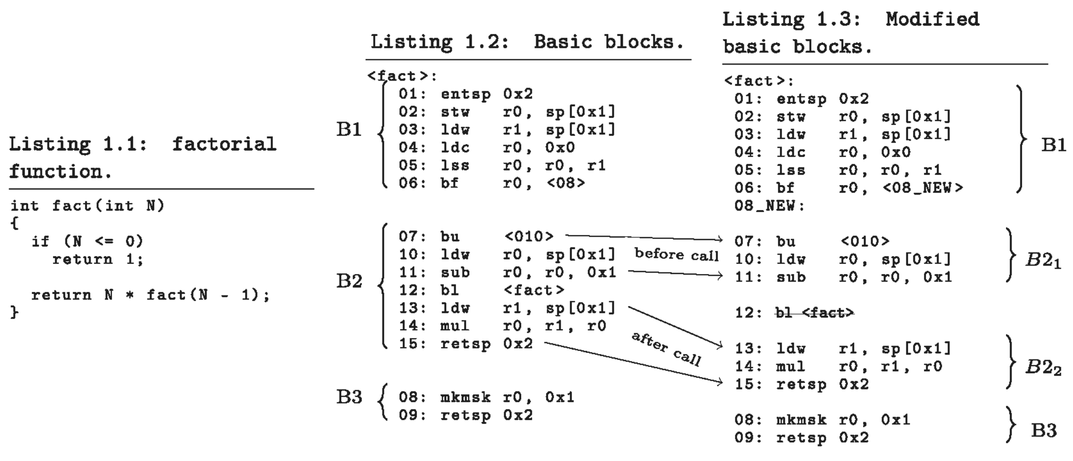
The first step of our energy bounds analysis is to determine upper and lower bounds on the energy consumption of each basic ('branchless') program block. We perform the modelling at this level rather than at the instruction level to cater for inter-instruction dependencies and to approximate non-conservative bounds.

Generating the basic blocks to be modelled. A *basic block* over an inter-procedural Control Flow Graph (CFG) is a maximal sequence of distinct instructions, S_1 through S_n, such that all instructions $S_k, 1 < k < n$ have exactly one in-edge and one out-edge (excluding call/return edges), S_1 has one out-edge and S_n has one in-edge. A basic block therefore has exactly one entry point at S_1 and one exit point at S_n. To divide a program into such *basic blocks*, the program is first compiled to the Instruction Set Architecture (ISA) representation, and then a data flow analysis of the CFG corresponding to the ISA representation is performed. The *basic blocks* are further modified so that they can be run/measured in isolation:

(1) A basic block with k function call instructions is divided into $k+1$ basic blocks without the function call instructions.
(2) A number of special ISA instructions (e.g. *return*, *call*) are omitted from the block. The cost of such instructions is measured separately and added to the cost of the block.
(3) Memory read/write instructions are abstracted to a fixed memory region available to each basic block to avoid memory violations.

An example of the modification 1 above is shown in Figure 3, Listing 1.2, which is an ISA representation of a recursive factorial program where the instructions are grouped together into 3 *basic blocks* $B1$, $B2$ and $B3$. Consider *basic block* $B2$. Since it has a (recursive) function call to *fact* at address 12, it is further divided into two blocks in Listing 1.3, so that the instructions before and after the function call form two blocks $B2_1$ and $B2_2$ respectively.

For each modified basic block, a set of input arguments is inferred. This set is used for an *individual* representation to drive the EA algorithm to maximize/minimize the energy consumption



Listing 1.1: factorial function.

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

  return N * fact(N - 1);
}
```

Listing 1.2: Basic blocks.

```
<fact>:
 01: entsp 0x2
 02: stw   r0, sp[0x1]
 03: ldw   r1, sp[0x1]
 04: ldc   r0, 0x0
 05: lss   r0, r0, r1
 06: bf    r0, <08>

 07: bu    <010>
 10: ldw   r0, sp[0x1]
 11: sub   r0, r0, 0x1
 12: bl    <fact>
 13: ldw   r1, sp[0x1]
 14: mul   r0, r1, r0
 15: retsp 0x2

 08: mkmsk r0, 0x1
 09: retsp 0x2
```

Listing 1.3: Modified basic blocks.

```
<fact>:
 01: entsp 0x2
 02: stw   r0, sp[0x1]
 03: ldw   r1, sp[0x1]
 04: ldc   r0, 0x0
 05: lss   r0, r0, r1
 06: bf    r0, <08_NEW>
 08_NEW:

 07: bu    <010>
 10: ldw   r0, sp[0x1]
 11: sub   r0, r0, 0x1

 12: ~~bl <fact>~~

 13: ldw   r1, sp[0x1]
 14: mul   r0, r1, r0
 15: retsp 0x2

 08: mkmsk r0, 0x1
 09: retsp 0x2
```

FIG. 3. Example: basic block modifications

of the block. For the entry block, the input arguments are derived from the signature of the function. The set $gen(B)$ characterizes the set of variables read without being previously defined in block B. It is defined as:

$$gen(b) = \bigcup_{k=1}^{n} \{v \mid v \in ref(k) \wedge \forall (j < k).v \notin def(j)\},$$

where $ref(n)$ and $def(n)$ denote the variables referred to and defined/updated at a node n in block b respectively.

For the basic blocks in Figure 3, Listing 1.2, the sets of input arguments are $gen(B1)$={r0}, $gen(B2_1)$={sp[0x1]}, $gen(B2_2)$={sp[0x1],r0} and $gen(B3) = \emptyset$.

The energy consumption of blocks $B2_1$ and $B2_2$ is maximized (minimized) by providing values to the input arguments to the blocks by using an EA. The energy consumption of $B2$ can be characterized as:

$$B2_e^A = B2_{1e}^A + B2_{2e}^A + bl_e^A,$$

where $B2_{1e}^A$, $B2_{2e}^A$ and bl_e^A denote the energy consumption of the $B2_1$, $B2_2$ blocks and the bl ISA instruction, with approximation A (where $A =$ upper or $A =$ lower).

EA for estimating the energy of a basic block. In the following we detail the most important aspects of the EA used for estimating the maximal (i.e. worst case) and minimal (i.e. best case) energy consumption of a basic block. The same EA is used for both cases, but the objective function is maximized or minimized for the former and latter case respectively.

Individual. The search space dimensions are the different input variables to the blocks. Our goal is to find the combination of input values which maximizes (minimizes) the energy of each block. Thus, an individual is simply an array of input values given in the order of their appearance in the block. In the initial population, the input values to an individual are randomly assigned to 32-bit numbers. In addition, some corner cases that are known to cause high (low) energy consumption for particular instructions are included.[1]

[1]For example all 1s for high energy consumption, or all 0s for low energy consumption as operands to a multiply ISA instruction.

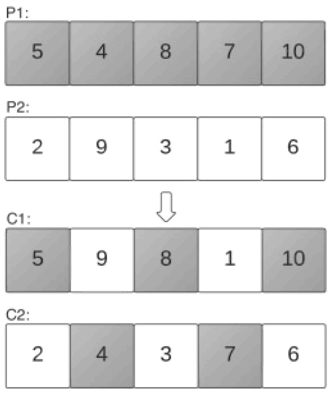


FIG. 4. Crossover

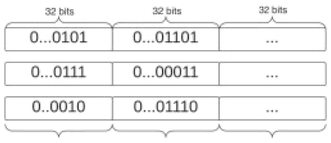


FIG. 5. Mutation

Crossover. We use an even-odd crossover, since it provides more variability than a standard n-point crossover. In our approach, the first child is created by taking the first element and every other one after it from one of the parents, e.g., the mother. The second element and every other one come from the other parent, i.e., the father. The second child is created in the opposite way. This operation is depicted in Figure 4, where $P1$ and $P2$ are the parents, and $C1$ and $C2$ are the resulting children.
Mutation. Since the (dynamic) energy consumption in digital circuits is mainly the result of bit flipping, the search space is explored by performing (random) bit flipping in the mutation operation as follows. For each gene (i.e. input value to the basic block):

(1) Randomly create a 32-bit integer, which will be used as a mask.
(2) Perform the XOR operation of this mask and the corresponding gene. This way, only the bits of the gene at positions where the value of the mask is 1 are flipped.

This operation is depicted in Figure 5, where the input values are given as binary numbers.
Objective function. The objective function that we want to maximize (or minimize) is the energy of a basic block, which is measured directly from the chip. The concrete settings for these experiments will be explained in Section 4.1.

In general, pipeline effects such as stalls (to resolve pipeline hazards), which depend on the state of the processor at the start of the execution of a basic block, can affect the estimated upper/lower bound on the energy consumption of such block. In our approach, intra-block pipeline effects are accounted for, since the dependencies among the instructions within a block are preserved. However, the inter-block pipeline effects need to be accounted for. These can be modelled in a conservative way by assuming a maximum stall penalty for the upper bound estimation of each block (e.g. by adding three cycles to the execution time of the block). Similarly, for the lower bound estimation a zero stall penalty can be used. To approximate this effect, in [6] the authors characterize each block through pairwise executions with all of its possible predecessors. Each basic block pair is characterized by executing it on an Instruction Set Simulation (ISS) to collect cycle counts.

The XMOS XS1 architecture used in our experiments does not have these pipeline effects by design, since exactly one instruction per thread is executed in a 4-stage pipeline (see Section 4.1 for more details).

3.3.2 Energy consumption of the program

Once the energy models of each basic block of the program are obtained, they are fed into an existing static analyser that takes into account the control flow of the program and infers safe upper/lower bounds on its energy consumption. Such analyser is a specialization of the generic resource analysis framework provided by CiaoPP [18] for programs written in the XC programming language [20] and running on the XMOS XS1-L architecture. We have also written the necessary code (i.e. assertions [9]) to feed this analyser with the block-level upper/lower bound energy model.

The generic resource analyser ensures that the inferred bounds are safe if it is fed with energy models providing also safe bounds. In [14] we performed a previous instantiation of such generic analyser by using the instruction-level energy model described in [10] that provided average energy values. As a result, the analysis inferred energy functions for the whole program that could possibly estimate values that are below the actual upper bound of the program.

The analysis is general enough to be applied to other programming languages and architectures (see [13, 14] for details) provided that an energy model for a particular architecture exists. It enables a programmer to symbolically bound the energy consumption of a program P on input data $\bar{x}$ without actually running $P(\bar{x})$, since it is based on an abstract domain that sets up a system of recursive cost equations that capture the energy consumption of the program as a function of the sizes of its input arguments $\bar{x}$. The transformation-based analysis framework of [13, 14] transforms 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).

Consider the example in Figure 3, Listing 1.2. The recursive cost equations that are set up characterize the energy consumption of the whole function *fact* using the approximation A for each block inferred by the EA:

$$fact_e^A(R0) \quad = \quad B1_e^A + fact\_aux_e^A(0 \leq R0, R0)$$

$$fact\_aux_e^A(B,R0) = \begin{cases} B2_e^A + fact_e^A(R0-1) & \text{if } B \text{ is } \texttt{true} \\ B3_e^A & \text{if } B \text{ is } \texttt{false}. \end{cases}$$

The cost of the *fact* function is captured by the equation $fact_e^A(R0)$ under approximation A (e.g. upper/lower) which in turn depends on $B1_e^A$ (i.e. the energy consumption of block $B1$) and the equation $fact\_aux_e^A$, which represents the branching originated from the last instruction of block *B1*. It captures the cost of blocks $B2$ and $B3$ based on the condition on the input size $R0$.

If we assume (for simplicity of exposition) that each basic block has unitary cost in terms of energy consumption, i.e., $Bi_e = 1$ for all i, we obtain the energy consumed by `fact` as a function of its input data size ($R0$): $fact_e(R0) = R0 + 1$.

The functions inferred by the static analysis are arithmetic functions (polynomial, exponential, logarithmic, etc.) that depend on input data sizes (natural numbers).

TABLE 1. Energy functions for 3 different pairs of voltage (V) / frequency (F, in MHz)

	(V,F) = (1.00,450)	**(V,F) = (0.87,400)**	**(V,F) = (0.80,350)**
fir(N)	$7.93\,N+24.5$	$5.36\,N+18.9$	$3.41\,N+15.2$
biquad(N)	$38\,N+12$	$22.6\,N+7.2$	$17.5\,N+5.2$

4 Experimental evaluation

4.1 Testing environment

XMOS Chips. As mentioned before, in our experiments we target the XMOS XS1-L architecture as a proof of concept. For building up the energy model and all the measurements in our experiments we use a board containing one core with 8 threads. All threads have their own register set and up to 4 instructions per thread can be buffered, which are scheduled in a way that minimizes simultaneous memory accesses by consecutive threads. 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 $N = max(4, \#Threads)$. As also mentioned before, DVFS is implemented at the chip level, which means that all the threads have the same voltage and frequency at a given point in time.

Task Set. We use two real-world programs for testing:

- `fir(N)`: Finite Impulse Response (FIR) filter. In essence, it computes the inner-product of two vectors: a vector of input samples, and a vector of coefficients.
- `biquad(N)`: Part of an equalizer implementation, which uses a cascade of Biquad filters. The energy consumed depends on `N`, the number of filters in the cascade, also known as banks.

These filters are often used in signal processing, where a certain level of accuracy loss can be permitted. This makes them good candidates for experimenting with the accuracy/energy trade-off. We have used four different FIR implementations, with different numbers of coefficients: 85, 97, 109 and 121. Furthermore, we have used four implementations of the Biquad program, with different numbers of banks: 5, 7, 10 and 14. We have tested our approach in scenarios with 32 tasks, each one corresponding to one of the above mentioned implementations. The tasks corresponding to the same implementation have different release times.

The energy consumed by the programs is inferred at compile time by the static analysis described in Section 3.3.2. The resulting energy functions from this analysis are given in Table 1 for the two benchmarks used. In the case of FIR, the parameter N is the number of coefficients, while in the case of the Biquad cascade, N is the number of banks. The analysis assumes that a single program is running on one thread on the XMOS chip, while all other threads are inactive. This means that only the first stage of the pipeline is occupied with an instruction, while the rest are empty, i.e., occupied with NOPs. In our implementation, the EA algorithm approximates the total energy of a schedule taking the sum of the energies of all the tasks running on different cores, i.e., threads, as we have seen in Section 3.2. However, in reality if all the threads are active and execute a program, each pipeline stage will contain an instruction from a different thread. For this reason, we can say that the estimation produced by the static analysis of the energy consumed by a set of tasks is an upper bound on the actual energy consumption. However, this estimation provides precise enough information for the EA to decide which schedule is better.

TABLE 2. Energy savings obtained with different levels of minimal acceptable accuracy.

Max. Avg. Error	Case 1: Avg. En.(mJ)	Case 2: Avg. En.(mJ)	Savings(%) Avg.	Savings(%) CI0.05
10^{-6}	0.487	0.721	16.18	0.93 – 31.42
$2 \cdot 10^{-6}$	0.461	0.597	18.21	3.54 – 32.87
$3 \cdot 10^{-6}$	0.434	0.666	31.04	13.72 – 48.37

4.2 Testing scenario

As already said, we have tested our approach on a scenario of 32 tasks, where each task implements either an FIR or a Biquad cascade. For the case of FIR, loop perforation takes out a few coefficients, while in the case of Biquad cascade, it takes out a few banks. All tasks have different release time. There are no task deadlines. However, we should bear in mind that in the case of DVFS it is not beneficial to scale down voltage and frequency indefinitely, since at some point the static power consumption becomes more significant than the dynamic part, resulting in an increment of the total energy consumption. The input signal to all tasks is a standardized set of input samples used for testing in the signal processing area.

4.3 Obtained results and discussion

The EA has been trained with the following parameters: a population of 200 individuals, evolved for 150 generations, with a crossover rate of 0.9, and a mutation rate of 0.9 — since mutation introduces loop perforation, a high rate is needed.

To illustrate the energy savings provided by loop perforation (referred to as *Case 1* in the following), we have trained another EA, where the objectives are to minimize energy and execution time, without the possibility of loop perforation (referred to as *Case 2* in the following). This algorithm has been trained with the same parameters given above. Since both algorithms are multiobjective, their result is a Pareto front of possible solutions with different trade-offs between the objectives.

In *Case 1* we have picked a solution with the smallest energy objective value, whose maximal deviation from the final result (accuracy) is below (above) a given threshold, while in *Case 2* we have chosen a solution with the smallest energy objective. The results are presented in Table 2, with the following columns:

- *Column 1:* Maximal acceptable average deviation (or equivalently, minimal acceptable level of accuracy) of the final result.
- *Column 2:* Average energy of the final schedule obtained for a set of experiments of *Case 1* by using static analysis, given in *mJ* (mili Joules).
- *Column 3:* The same as *Column 2* but for *Case 2*.
- *Column 4:* Percentage of savings obtained, calculated as $\frac{Column3 - Column2}{Column3} \cdot 100$.
- *Column 5*: Statistics of the experiments expressed as a 0.05 confidence interval, i.e., 95% certainty that the final result will belong to this interval.

As we can observe, the energy savings that can be obtained with loop perforation are significant and range from 3% to 40% in the different experiments, even with a small permitted level of deviation. The savings are proportionally the same, irrespective of whether we use average or upper-bound energy models. As expected, Figure 6 shows that if we increase the accepted level of average deviation, the energy savings also increase. However, the relationship between the accuracy and the

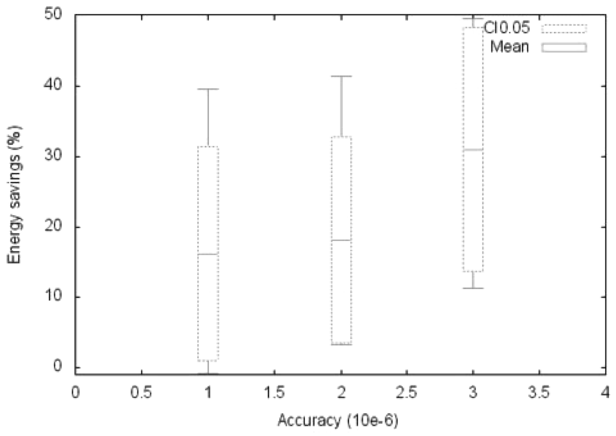


FIG. 6. Energy savings for different accuracy levels

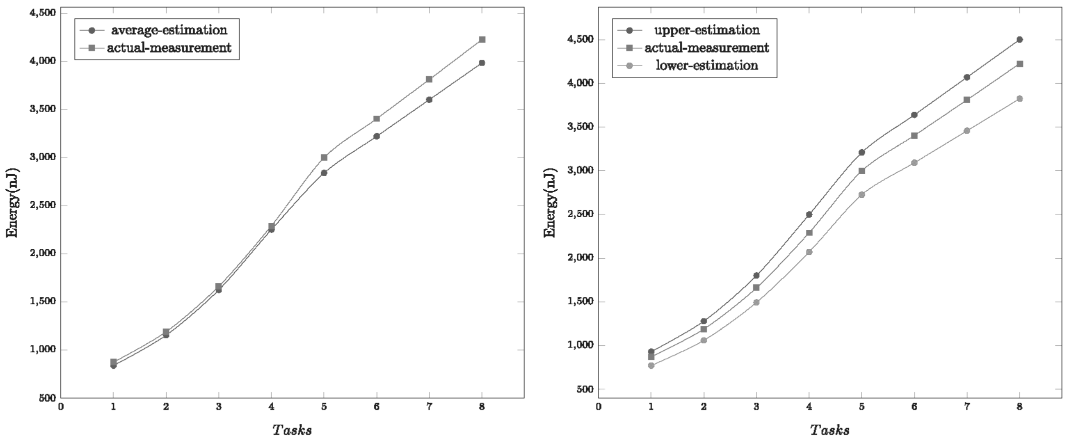


FIG. 7. Comparing the actual energy of a schedule with estimations using average models (left) and upper/lower bound models (right)

energy savings depends on the application: some applications can preserve acceptable accuracy by skipping more loop iterations (and hence achieve bigger energy savings) than others.

Since the acceptable level of deviation is small, and the number of iterations performed by FIR is bigger than the one corresponding to Biquad cascade, we can observe that in the final result tasks that perform FIR can skip more iterations than the ones performing Biquad, which can skip one iteration at most.

In Figure 7, a random schedule of 8 tasks (comprising 4 tasks of each of Biquad and FIR filters) executing sequentially on a single thread, is selected for measurement on the hardware. The graph on the left compares the actual measurement of the schedule against the estimated energy consumption using average energy models, while the graph on the right compares the actual measurement against estimations using the approach described in Section 3.3. The latter estimations are safe, in the sense that the energy consumption inferred using upper (resp. lower) bound models always over- (resp.

under-) approximate the actual measurement. In contrast, the former estimations slightly under-estimate the actual energy measurement.

5 Conclusions

In this work, we have presented an approach for energy-efficient scheduling in multicore environments, adapted to multicore XMOS processors, where significant additional energy can be saved if a certain level of accuracy reduction in the final result is allowed. This gradual accuracy reduction is achieved by using the loop perforation technique. To certify/verify that a schedule meets an energy budget, safe bounds are needed (both upper and lower bounds). Since our previous estimations using an average energy model do not meet such condition, we have used a parametric approach that estimates safe (and tight) energy bounds that can be used for energy verification in practice. Our experimental results show that, even with small acceptable levels of deviation in the output, significant energy savings can be obtained.

Acknowledgements

This research has received funding from EU FP7 agreement no 318337, ENTRA, Spanish MINECO TIN2012-39391 *StrongSoft* and TIN2015-67522-C3-1-R *TRACES* projects, and the Madrid M141047003 *N-GREENS* program.

References

[1] J. Bader and E. Zitzler. Hype: an algorithm for fast hypervolume-based many-objective optimization. *Evolutionary Computation*, **19**, 45–76, 2011.

[2] Z. Banković, U. Liqat and P. López-García. A practical approach for energy efficient scheduling in multicore environments by combining evolutionary and YDS algorithms with faster energy estimation. In *The 11th International Conference on Artificial Intelligence Applications and Innovations (AIAI'15)*, Vol. 458 of *IFIP Advances in Information and Communication Technology*, pp. 478–493. Springer, 2015.

[3] Z. Banković, U. Liqat and P. López-García. Trading-off accuracy vs. energy in multicore processors via evolutionary algorithms combining loop perforation and static analysis-based scheduling. In *Hybrid Artificial Intelligent Systems (HAIS 2015)*, Vol. 9121 of *LNCS*, pp. 690–701. Springer International Publishing, 2015.

[4] Z. Banković and P. Lopez-Garcia. Stochastic vs. deterministic evolutionary algorithm-based allocation and scheduling for XMOS chips. *Neurocomputing*, **150**, 82–89, 2015.

[5] N. Beume, B. Naujoks and M. Emmerich. Sms-emoa: multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, **181**, 1653–1669, 2007.

[6] S. Chakravarty, Z. Zhao and A. Gerstlauer. Automated, retargetable back-annotation for host compiled performance and power modeling. In *Proceedings of CODES+ISSS '13*, pp. 36:1–36:10. IEEE Press, 2013.

[7] K. Deb and H. Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, **18**, 577–601, 2014.

[8] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan. A fast elitist multi-objective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, **6**, 182–197, 2000.

[9] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. F. Morales and G. Puebla. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming*, **12**, 219–252.

[10] S. Kerrison and K. Eder. Energy modeling of software for a hardware multithreaded embedded microprocessor. *ACM Transactions on Embedded Computing Systems*, **14**, 1–25, 2015.

[11] P. R. Kumar and S. Palani. A dynamic voltage scaling with single power supply and varying speed factor for multiprocessor system using genetic algorithm. In *Proceedings of PRIME'12*, pp. 342–346. IEEE, 2012.

[12] U. Liqat, Z. Banković, P. Lopez-Garcia and M. V. Hermenegildo. Inferring energy bounds statically by evolutionary analysis of basic blocks. In *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2016)*, arXiv:1601.02800, 2016.

[13] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher and K. Eder. Inferring parametric energy consumption functions at different software levels: ISA vs. LLVM IR. In *Proceedings of FOPARA*, Vol. 9964 of *LNCS*, pp. 81–100. Springer, 2016.

[14] 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*, Vol. 8901 of *LNCS*, pp. 72–90. Springer, 2014.

[15] H.-L. Liu, F. Gu and Q. Zhang. Decomposition of a multiobjective optimization problem into a number of simple multiobjective subproblems. *IEEE Transactions on Evolutionary Computation*, **18**, 450–455, 2014.

[16] M.-S. Mezmaz, Y. Kessaci, Y. C. Lee, N. Melab, E.-G. Talbi, A. Y. Zomaya and D. Tuyttens. A parallel island-based hybrid genetic algorithm for precedence-constrained applications to minimize energy consumption and makespan. In *Proceedings of GRID'10*, pp. 274–281. IEEE, 2010.

[17] M. Mezmaz, Y. C. Lee, N. Melab, E. Talbi and A. Y. Zomaya. A bi-objective hybrid genetic algorithm to minimize energy consumption and makespan for precedence-constrained applications using dynamic voltage scaling. In *Proceedings of CEC'10*, pp. 1–8. IEEE, 2010.

[18] 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**, 739–754, 2014.

[19] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of FSE'11*, pp. 124–134. ACM, 2011.

[20] D. Watt. *Programming XC on XMOS Devices*. XMOS Limited, 2009.

[21] S. Yassa, R. Chelouah and B. Granado. Multi-objective approach for energy-aware workflow scheduling in cloud computing environments. *The Scientific World Journal*. Article ID: 350934, 2013.

[22] G. Ye, R. Rao and M. Li. A multiobjective resources scheduling approach based on genetic algorithms in grid environment. In *Proceedings of GCCW '06*, pp. 504–509. IEEE, 2006.

[23] J. Yu, M. Kirley and R. Buyya. Multi-objective planning for workflow execution on grids. In *Proceedings of GRID'07*, pp. 10–17. IEEE, 2007.