**ENTRA**

**318337**

**Whole-Systems ENergy TRAnsparency**

# Energy Optimization: Advanced Techniques (including SW demo)

| | |
|---|---|
| Deliverable number: | D4.2 |
| Work package: | Optimization (WP4) |
| Delivery date: | 1 October 2015 (36 months) |
| Actual date: | 1 March 2016 |
| Nature: | Prototype |
| Dissemination level: | PU |
| Lead beneficiary: | Roskilde University |
| Partners contributed: | Roskilde University, University of Bristol, IMDEA Software Institute, XMOS Limited |

**Project funded by the European Union under the Seventh Framework Programme, FP7-ICT-2011-8 FET Proactive call.**

**Short description:**

This deliverable presents results relating to tools and methods supporting energy optimisation. Both static and dynamic techniques are covered.

The deliverable includes the following attachments:

- D4.2.1: *Stochastic vs. Deterministic Evolutionary Algorithm-based Allocation and Scheduling for XMOS Chips*. Neurocomputing, Vol. 150, pages 82–89, Elsevier, February 2015.

- D4.2.2: *Energy Efficient Allocation and Scheduling for DVFS-enabled Multicore Environments using a Multiobjective Evolutionary Algorithm*. Genetic and Evolutionary Computation Conference (GECCO 2015), pages 1353–1354, ACM, 2015.

- D4.2.3: *Trading-off Accuracy vs. Energy in Multicore Processors via Evolutionary Algorithms Combining Loop Perforation and Static Analysis-based Scheduling*. Hybrid Artificial Intelligent Systems (HAIS 2015), Lecture Notes in Computer Science, Vol. 9121, pages 690–701, Springer International Publishing, 2015.

- D4.2.4: *Improved Energy-aware Stochastic Scheduling based on Evolutionary Algorithms via Copula-based Modeling of Task Dependences*. International Conference on Soft Computing Models in Industrial and Environmental Applications (SOCO 2015), Advances in Intelligent Systems and Computing, Vol. 368, pages 153–163, Springer International Publishing, 2015.

- D4.2.5: *A Practical Approach for Energy Efficient Scheduling in Multicore Environments by combining Evolutionary and YDS Algorithms with Faster Energy Estimation*. The 11th International Conference on Artificial Intelligence Applications and Innovations (AIAI'15), IFIP Advances in Information and Communication Technology, Vol. 458, pages 478–493, Springer, 2015.

- D4.2.6: *Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable XMOS Chips*. Hybrid Artificial Intelligent Systems (HAIS 2013), Lecture Notes in Computer Science, Vol. 8073, pages 401–410, Springer, 2013.

- D4.2.7: *An Energy-Aware Programming Approach for Mobile Application Development Guided by a Fine-Grained Energy Model*. Technical report, Roskilde University, February 2016. to be submitted for publication.

# Contents

## 1 Introduction

This deliverable presents results relating to tools and methods supporting energy optimisation. There are four main sections to the report. The global optimiser and the dual-issue processor constitute the main "prototype" contained in the deliverable.

- In Section 2 we describe the global optimiser included as of Release 14 of the XMOS tools; this is a stage in the XC build process where whole-program analysis is performed in order to improve energy behaviour.

- In Section 3 we give an overview of the XMOS XS2 architecture supporting dual-issue, that is, the ability to execute two instructions simultaneously during one clock cycle. A key motivation for this architecture is that it cuts the energy overhead associated with executing instructions, involving distributing clocks, maintaining program counters, resources, etc.

- In Section 4 we describe a case study in static energy optimisation of source code based on energy transparency. The case study deals with Android app code, for which detailed energy accounting is performed based on a source code energy model. This leads to manual optimisations of the most energy-consuming parts of the code, which save up to 50% of energy in some use scenarios. This work is being further developed and being incorporated into tools supporting energy-aware mobile app development.

- In Section 5 we present a summary of a body of theoretical and experimental work on *dynamic* energy optimisation. Dynamic energy optimisations aim to use an energy model to make intelligent run-time decisions on task allocation in multicore multithreaded environments with the possibility of voltage and frequency scaling, with given constraints and requirements on performance. This work relies heavily on ENTRA tools for energy modelling and analysis to provide the information needed for the optimisation algorithms.

## 2 Global optimiser

As of release 14, the XMOS tools incorporate a global optimiser; a stage in the build process where whole-program analysis is performed in order to improve energy behaviour.

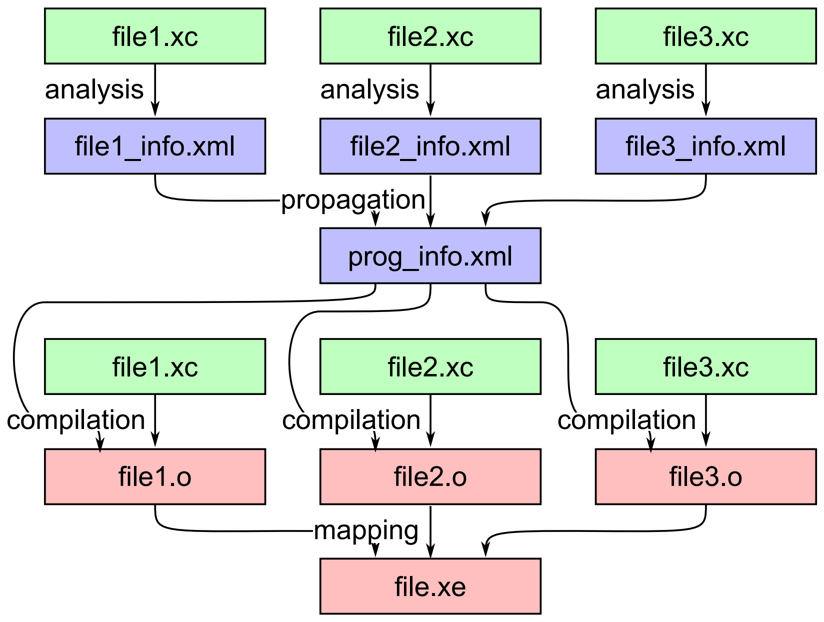


Figure 1: Global optimiser structure

## 2.1 Introduction

The global optimiser is a stage in the build process that analyses the call graph of an XCORE program, and performs global constant propagation throughout the program, enabling later stages in the compiler to generate efficient code.

The reason why a special phase was invented rather than taking a more traditional approach is that:

- We need intra-modular analysis, going across boundaries of source files
- We need source-level analysis, because once we have reached the LLVM intermediary level we have lost too much semantic information

This leads us to a tool that works as shown in Figure 1. It comprises three stages:

- A tool to analyse a single source file, and to write the results of that analysis into an XML file that identifies function definitions, function calls, and constants. This tool is built into the front end, and a flag on the front-end emits the XML file instead of performing a compilation.

- A tool that takes all the XML source files, and based on this provide specialisations for (each of) the source functions. This is a new tool, `xpca`, it generates a single XML file that contains all the information.
- Based on the output of xpca, modules are recompiled. The final compilation stage uses the program information file to specialise specific functions.

## 2.2 Rationale

There are two reasons to implement a global optimiser.

- The first reason is that it can improve the energy profile of existing code
- The second reason is that it enables software engineers to design their software in a more modern, concise and consistent way.

The latter point is especially relevant; it is very easy to write energy efficient software on a small scale. It is hard to properly engineer efficient software in such a way that it can be reused, maintained, and read without specialist knowledge.

Interfaces (see Deliverable D1.2) were introduced to aid in software engineering; but their generic implementation is not efficient. It is the combination of interfaces and the global optimiser which creates a potent tool that generates code that is more efficient than naively written code.

Depending on how the interface is used, the global optimiser may recover some or all inefficiencies caused by the interface itself. As an example, consider the I2C interface illustrated in D1.2:

```
void i2c_master(server interface i2c_master_if c[n], size_t n,
                port p_scl, port p_sda, unsigned kb_per_sec) {
    unsigned bit_time = (XS1_TIMER_MHZ * 1000) / kb_per_sec;
    unsigned locked_client = -1;
    p_scl :> void;
    p_sda :> void;
    while (1) {
        select {
        case (size_t i =0; i < n; i++)
          (n==1 || locked_client == -1 || i == locked_client) =>
            c[i].read(uint8_t device, uint8_t buf[m], size_t m,
                 int send_stop_bit) -> i2c_res_t result:
            ...
            locked_client = send_stop_bit ? -1 : i;
            result = (ack == 0) ? I2C_ACK : I2C_NACK;
            break;

        case c[int i].send_stop_bit(void):
            ...
            locked_client = -1;
            break;
        ...
        }
    }
}
```

This interface works on `n` clients, where `n` is chosen by the application designer. A common choice is `1` client only, in which case the code below will optimise to remove all the case-sequences, remove the guards, remove the `locked_client` variable, and all array bound checks on the `c` array.

Further optimisations happen because not all parts of the interface may be used. Say that the client only uses the `read()` call, but never writes data, and it always sends the implicit stop bit, never calling the `send_stop_bit()` call. In that case, the code for these will be removed. This will not improve energy efficiency, but it will reduce the memory footprint of the application, which agains supports software engineering and avoids designers having to make bespoke versions of interfaces.

An evaluation of the global optimiser is part of Deliverable D6.2.

Aside from optimising the interfaces, the global optimiser enables software modules to be configured in a way that does not require a pre-processor. If we take a UART as an example, then there is two ways to configure the UART. Configuration using `#define` lines requires those to be included in the right files through a user defined include-file that is included by the module; `uart_conf.h` would contain:

```
#define UART_PARITY       0     // Disable parity
#define UART_STOP_BITS    1     // One stop bit
#define UART_BAUD_RATE    9600  // Speed to run at
```

And then there would be some magic that causes all UART source files to include this configuration header file. Alternatively, the UART module would allow configuration through the instantiation of the thread. The `uart.h` include file contains the definition of the uart thread:

```
extern uart_thread(in parity, int stop_bits, int baud_rate);
```

And the main program invokes the thread with the appropriate parameters:

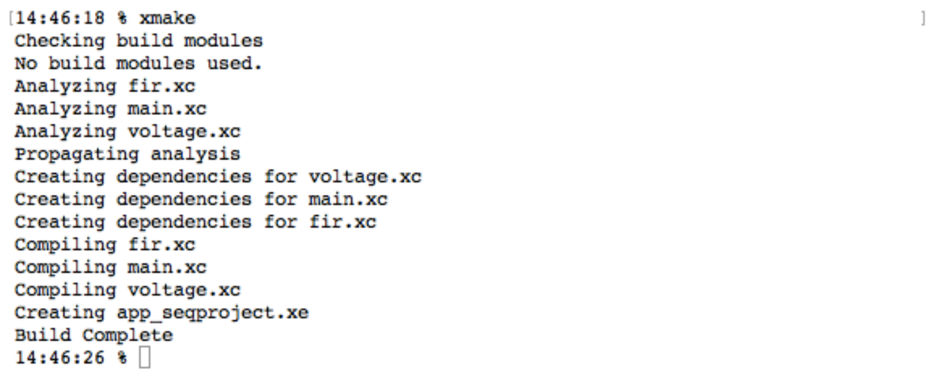
```
#incude <uart.h>

main() {
  par {
     uart_thread(0, 1, 9600);
  }
}
```

With the global optimiser, these two options are now identical in terms of energy usage and speed; however, the latter option is preferable from a software perspective as there is no magic involved in importing the defines into the library

## 2.3 Invocation from tools

The XMOS tool chain will automatically invoke all the tools necessary. Since the XMOS tool chain contains the full build system (including the dependency checker), it has full control anyway, and it is straightforward to call the analysers and global optimisers in a transparent manner. This is shown in Figure 2.



```
14:46:18 % xmake
Checking build modules
No build modules used.
Analyzing fir.xc
Analyzing main.xc
Analyzing voltage.xc
Propagating analysis
Creating dependencies for voltage.xc
Creating dependencies for main.xc
Creating dependencies for fir.xc
Compiling fir.xc
Compiling main.xc
Compiling voltage.xc
Creating app_seqproject.xe
Build Complete
14:46:26 %
```

Figure 2: XMAKE screenshot

One can see that the build process comprsies five steps, three of which (steps 1, 2, and 4) relate to the global optimiser. Step one analyses all the source files that have been changed since the last build. Step two propagates the analysis, incorporating the analyses of source files that have not been changed. Step three creates dependencies. Step four performs the modular compilation, using the propagated global optimisation analysis. Step five builds the final object.

Users of the XMOS tools release 14 (and beyond) are using globally optimised code by default.

# 3 Dual-issue processing

## 3.1 Introduction

Of the total energy consumed by a device when executing a program, a large fraction of this time is related to overhead involved in executing a program; that may involve distributing clocks, maintaining program counters, resources, etc.

Given this observation, we have developed the XS2 architecture that supports dual issue: executing two instructions simultaneously during one clock cycle.

Executing dual issue, means that we can execute the same code in fewer clock cycles, consuming less power due to a cut in the overhead.

## 3.2 Dual issue design

The dual issue design of the XS2 architecture pivots around the two *lanes* that are used to execute instructions. The lanes are not symmetrical, for that would have added an unreasonable amount of logic and complication. Instead, the lanes are specialised as follows:

**IO Lane** The IO Lane can be used to execute all resource instructions, instructions that modify the thread state (such as the status register), and all basic arithmetic operations.

**Memory lane** The Memory Lane can be used to execute all memory operations (load, store, and branch operations, including subroutine call and return), and all basic arithmetic operations.

Hence, resource instructions can only execute in the left pipeline, and memory instructions can only execute in the right pipeline:

- It is not possible to dual issue two resource operations. Allowing this would have complicated the resource logic to deal with multiple simultaneous operations and would have added to the energy profile of the processor.
- It is not possible to dual issue two memory operations. Allowing that would have forced another port on the memory.
- It is possible to dual issue two basic operations (eg, add and add).
- It is possible to dual issue basic operations with either IO, or memory.
- It is possible to dual issue a memory operation with a resource operation.

- Two operations should never write to the same register.

Many of the inner loop that XCORE programs exhibit show a mixture of memory and resource operations, or memory and arithmetic operations. For example, a loop to input or output a block of data uses a resource operation and a load/store operation. They depend on each other, but by unrolling the loop once they can be dual issued. Similarly, memory and arithmetic are a normal combination.

In addition to the simple arithmetic operations there are complex operations such as multiply, divide, long add, CRC, etc. All of these instructions are rarer, may be associated with a large number of gates, and may have many input and output operands. They are encoded in a long 32-bit encoding, and are executed in single issue. All dual issuable instructions are encoded in 16-bits and can be executed side by side.

## 3.3 Extra instructions to support dual issue

A number of instructions have been added to the instruction set to further improve energy performance, and to partly counteract the restrictions above:

- Dual load and store instructions have been added. They are long instructions that fetch (store) a pair of words from an address that is double word aligned. A full dual issue machine would enable an arbitrary pair of words to be loaded; being able to load a pair of subsequent words is a useful intermediary for many uses. For example, complex numbers, coefficients of filters, subsequent items in a queue, or subsequent items in a datablock can all be load/stored at twice the speed using these instructions.

- Similarly, dual load and store instructions for the stack have been added that facilitate more efficient function entry and exit blocks.

- A few extra arithmetic instructions have been added to encode sequences of dependent instructions. In particular, saturating arithmetic, extracting the result, and extracting/inserting words of data are supported. They are difficult to perform efficiently in dual issue code as there are data dependencies that make dual issue hard.

- A DUALENTSP instruction is added that marks the entrance to a piece of dual issue code. This way, each function can be written in either dual or single issue; and code that has to be energy efficient can be made to run in dual issue, whilst code that has to be memory efficient can be written in single issue.

Finally, a dedicated NOP instruction has been added. Dual issue slots that cannot be used must be filled by a NOP instruction. Traditionally, the XS1 architecture supported a large number of instructions that are effectively NOP instructions, such `or r0,r0,r0`, `or r1,r1,r1`, `add r11,r11,0`. However, all these instructions always write into a register, and it would be problematic if the other lane wrote in the same register. It also uses less power.

## 3.4 Example dual issue

As an example of how dual issue with the other instructions work out, we show the inner loop of an FFT on XS2 below:

```
innerLoop1:
         ldd   r3, r6, r4[0]
         ashr  r6, r6, 1
         ashr  r3, r3, 1
         ldd   r2, r5, r4[r9]
         ldd  r10, r7, sp[10]
         maccs r8, r7, r5, r1
         maccs r8, r7, r2, r0
        {ldc r7, 0           ; neg r5, r5      }
         maccs r7, r10, r5, r0
         maccs r7, r10, r2, r1
        {add  r6, r6, r8     ; sub r8, r6, r8  }   // Cross use
        {add  r3, r3, r7     ; sub r7, r3, r7  }   // Cross use
         std   r3, r6, r4[0]
         std   r7, r8, r4[r9]
        {ldw  r6, sp[16]     ; sub r4, r4, r11 }
        {lsu  r8, r4, r6     ; nop             }
        {bf   r8, innerLoop1 ; nop             }
```

- This loop comprises 17 issue slots; executing 21 instructions. Superficially, this is a disappointing issue rate of 1.23 instructions/thread cycle. However, the loop also contains five double loads and stores, that are actually shortcuts for dual-issued loads and stores. Hence, a fair metric is that it executes 26 instructions in 17 issue slots, giving an issue rate of 1.52 instructions/thread cycle. This reduces power by a third.
- The code has an interesting feature in issue slots 11 and 12 (marked `Cross use`). In these slots, the destination operands in both lanes, are also source operands in both lanes.

Each instruction uses R6 and R8, and the left lane writes into R6 whereas the right lane writes into R8. This piece of code cannot be serialised into single issue, for it would either overwrite R6 or R8 before the register is dead. Indeed, executing this code in single issue requires a register to be spilled, which would add a total of four instructions to the loop. This brings the comparable single issue code to take 30 thread cycles; compared to 17 thread cycles for the dual issue code, a reduction of a factor of 1.75.

- The final difference stems from the new pre-fetch mechanism. To keep the dual issue pipeline going the instruction buffers had to be extended, and more data is kept in them. Due to this, the total number of thread cycles saved compared to the old single-issue architecture is close to a factor of 2.

The improvement in issue rate of nearly a factor of 2 is offset by slightly higher power consumption due to the extra lane. It is very difficult to perform a precise comparison because various other parts of the processor were improved (for energy efficiency), however, we think that overall on average we save between a factor of 1.5 and 2.

## 4 Energy-accounting for Android app energy optimisation

Energy transparency can be applied in software optimisation in many ways. In this section we describe a case study in energy-aware program optimisation in Android app development. The approach is based on manual optimisations guided by a fine-grained energy model of Android source code. The overall approach is as follows.

- We utilize the methodology described in [LG15] to construct the operation-based source-code-level *energy model*, which is achieved by analyzing the data produced in a range of well-designed execution cases.
- The model generates *energy accounting* at operation and block level, which captures the energy characteristics of the code.
- We *focus manual optimisation* efforts on the most costly blocks, where we refactor the code to remove, reduce or replace the expensive operations, meanwhile maintaining its logical consistency with the original code.

The experimental result shows that our approach is able to save from 6.4% to 50.2% of the overall energy consumption depending on different scenarios.

Table 1: Examples of Energy Operations

| Operation | *Identified where:* |
|---|---|
| Method Invocation | *one method is called* |
| Parameter_Object | *Object is one parameter of the method* |
| Return_Object | *the method returns an Object* |
| Addition_int_int | *addition's operands are integers* |
| Multi_float_float | *multiplication's operands are floats* |
| Increment | *symbol "++" appears in code* |
| And | *symbol "&&" appears in code* |
| Less_int_float | *"<"'s operands are integer and float* |
| Equal_Object_null | *"=="'s operands are Object and null* |
| Declaration_int | *one integer is declared* |
| Assign_Object_null | *assignment's operands are Object and null* |
| Assign_char[]_char[] | *assignment's operands are arrays of chars* |
| Array Reference | *one array element is referred* |
| Block Goto | *the code execution goes to a new block* |

"Energy operations" are the basic units in the source code that consume energy; all statements, blocks and methods are made up of a energy operations. In our experiment, we have 120 operations. This a fine-grained compare to other approaches to modelling energy at source code level. Energy information at the level of source lines or methods is useful; however, information at source line level could not distinguish energy consumption of two operations in the same source line, for example, which can be captured by modelling energy operations.

## 4.1 Case study outline

Our target platform is an Android development board with two ARM quad-core CPUs, and the source code in our study is a game engine used in games, demos and other interactive applications. We evaluate the approach in three game scenarios, and the experimental result shows that it can save varying amounts of energy in different scenarios. The full details of the architectural setup, the design of execution cases for energy modelling and the model construction can be found in Attachment D4.2.7. We examine three different typical scenarios in user interaction during game-playing, based on which we are able to capture the energy characteristics and optimise the source code in three different scenarios, namely `Click & Move`, `Orbit` and `Waves`.

Table 2: Examples of Library Functions

| Class | Function |
|---|---|
| ArrayList | *add, get, size, isEmpty, remove* |
| GL10 | *glBindTexture, glDisableClientState*<br>*glDrawElements, glEnableClientState*<br>*glMultMatrixf, glTexCoordPointer*<br>*glPopMatrix, glPushMatrix*<br>*glTexParameterx, glVertexPointer* |
| Math | max, pow, sqrt, random |
| FloatBuffer | *position, put* |

### 4.1.1 Basic Energy Operations

Energy operations are identified directly from source code. The enumeration of the operations is inspired by Java semantics [BR15], which specifies the operational meaning, or behaviour, of the Java language, which is the target language in the experiment. We intuitively identify semantic operations that perform operations on the state and may be energy-consuming, and let them be our energy operations. Operations that turn out to have little or no energy effect will automatically be identified by the regression analysis in the later stage of the energy modelling. Table 1 lists 14 representative operations out of a total of 120 in the experiment. They include arithmetic calculations like *Multi_float_float*, *Addition_int_int*, in which operands types are explicit, as well as *Increment* whose operand is implicitly an integer. Boolean operations and comparisons, such as *And*, *Less_int_float* and *Equal_Object_null* also form one major part. *Method Invocation* and *Block Goto* are important for the control flow which plays a key role in the execution of the code. Assignments and *Array Reference* will unexpectedly take a significant amount of the application's energy consumption.

The application also employs a diversity of library functions that may be written in different languages and at lower levels of the software stack. On the other hand, usually a limited number (67 in the experiment) of library functions are frequently called in one application. So we treat them as basic modelling units. The examples of highly-used library functions in the experiment are shown in Table 2. For instance, the functions in the class of *GL10* are responsible for graphic computing.

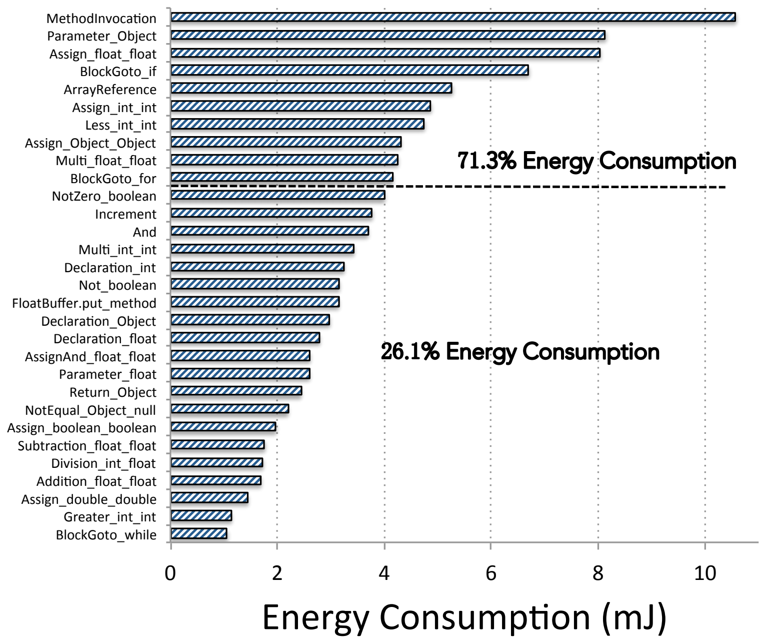


Figure 3: The top 30 energy consuming operations in `Click & Move` scenario.

### 4.1.2 Energy Model

An energy model is constructed (see Attachment 4.2.X for details) yielding an energy value for each energy operation. Using this model, together with logs from the test cases used to build the energy model, we can discover many useful pieces of information about the game-engine code. The energy model of app source code based on energy operations facilitates comprehensive *energy accounting* at different levels of granularity and from various viewpoints. For example, we can identify the most energy-expensive operations and source code blocks.

## 4.2 The `Click & Move` Scenario

We first consider optimising the energy in the `Click & Move` Scenario. We use energy accounting at operation and block level, according to which we improve the most costly blocks by removing, reducing or replacing the most expensive operations. Later in Section 4.3 and Section 4.4, when we talk about the `Orbit` and `Waves` scenarios, we will briefly introduce the energy characteristics of the code and use larger part for the code improvements.

**Operation Level.** Figure 3 shows the top 30 energy consuming operations, which are ranked by their single-execution energy costs. The section marked “71.3% Energy Consumption” shows the percentage of the sum of costs of the top 10 operations in the total cost, considering their different numbers of executions in the `Click & Move` scenario, while "26.1% Energy Consumption" means the percentage of operations from 11th to 30th. The percentages indicate that the energy-usage of the code is largely determined by a relatively small number of operations, because these operations are both frequently used and expensive themselves. The 30 operations out of 187 (including library functions) take up 97.4% of the whole cost of the code, in which the top 10 consumes the major part with a percentage of 71.3%.

Usually, it is supposed that the complex arithmetic operations, such as multiplications and divisions, should be the most costly. However, the result shows that in terms of source code operations, *Method Invocation* ranks the highest. This is because *Method Invocation* involves a sequence of operations, such as storing the return address and managing the stack frame, while instance methods are always implicitly passed a "this" reference as their first parameter.

Unexpectedly, only one arithmetic operation, *Multi_float_float*, is a member of the top 10, and there are only six arithmetic operations in the top 30. They together cost only 6.1% of the overall energy consumption of the application, which is contrary to our intuition.

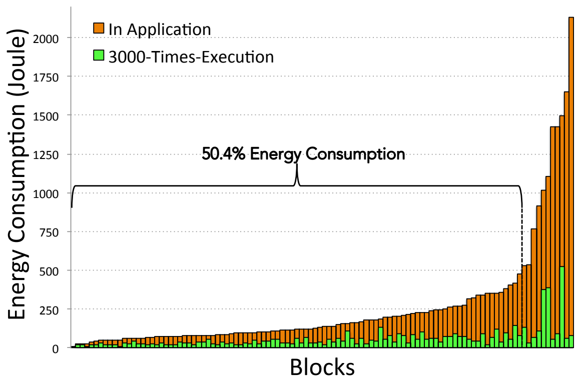
We will see later in block-level energy accounting that assignments, comparisons and *Array Reference* play significant roles in the overall energy consumption. This is not only because they are frequently used, but also because they are costly in themselves, as shown in Figure 3.

*Block Goto* operations are expensive as well. Based on the types of conditionals and loops where "Block Goto" occurs, they are classified into *BlockGoto_if*, *BlockGoto_for* and *BlockGoto_while*. The result shows that they cost different amounts of energy as operations themselves, respectively 6.7 mJ, 4.1 mJ, 1.1 mJ. And together with *Method Invocation*, they take up 37.6% of the total energy consumption of the application.

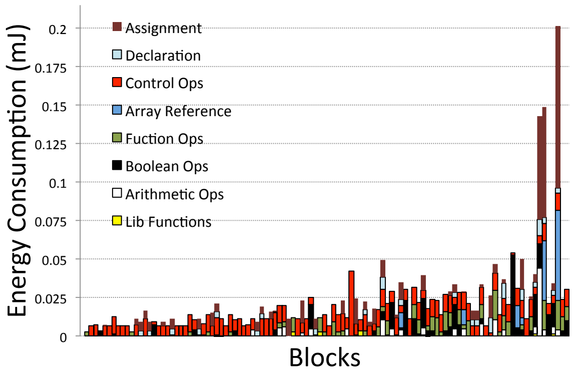
**Block Level.** In the execution cases, we have 108 active blocks with a wide diversity of energy usage. As shown in Figure 4a, "In Application" here means running the `Click & Move` scenario with the full set of blocks. The costs of blocks "In Application" are plotted as orange bars. Note that, blocks here obviously have distinct execution times. The cost of a fixed number (3000) of executions of one block are calculated by multiplying its single-execution cost by 3000. This could help us compare the single-execution costs of different blocks. The costs of blocks at "3000-Times-Execution" are plotted as green bars.

Similar to energy distribution on operations, a small number (11 blocks) of all the blocks uses up nearly half of the entire cost, which indicates that putting efforts on optimising a small group of blocks can achieve significant energy-saving.

There are two factors that make one block costly "In Application". The first factor is a large number of executions. For example, the most costly block "In Application" (the rightmost



(a) Block costs ”In Application” and at ”3000-Times-Execution”.



(b) Energy proportions of different kinds of operations in blocks.

Figure 4: Energy distribution in `Click & Move`. Blocks are sorted by the order of their run-time energy costs ”In Application”.

orange bar in Figure 4a) has a large number of execution times. This block takes only 30.6 mJ for single-execution but 2128.6 J when running “In Application”. The second factor is the energy consumption of the block itself. For example, the three prominent green bars in Figure 4a, whose single-execution costs are 201.5 mJ, 146.9 mJ and 142.8 mJ. We will later zoom in these three blocks to see which operations contribute to their energy costs.

We can further observe the energy proportions of operations in each block in Figure 4b. To illustrate, operations are grouped into eight classes. Specifically, the “Block Goto” operations and *Method Invocation* are gathered in *Control Ops*; the parameter passing and the value returns of methods are in *Function Ops*; the comparisons and Booleans are in *Boolean Ops*; all the arithmetic computations are in *Arithmetic Ops*; all the library functions are in *Lib Functions*.

Most of the blocks cost less than 25 mJ for single-execution. In these blocks, *Control Ops* occupy the major part of the energy consumption, in contrast, *Arithmetic Ops* only take a tiny proportion.

For those three most prominent blocks, assignments and *Array Reference* are the biggest energy consumers. Furthermore one of the three blocks has the largest proportion of *Arithmetic Ops* among all the blocks.

The most expensive block “In Application” consists of three even parts: *Control Ops*, *Function Ops* and *Boolean Ops*. This block is the main entrance of the game engine to draw and display frames, so its works are conditional judgments and method invocations.

Table 3: The top 10 most costly blocks in `Click & Move`.

| Block ID | #Executions | Energy Cost (J) |
|---|---|---|
| CCNode.visit() | 19462 | 2128.6 |
| CCNode.transform() | 18903 | 1648.4 |
| CCTextureAtlas.putVertex() | 2119 | 1494.4 |
| CCNode.visit().if_4.for_1 | 16880 | 1426.8 |
| CCNode.transform().if_1 | 19664 | 1426.3 |
| CCTextureAtlas.putTexCoords() | 2120 | 1107.8 |
| CCAtlas.updateValues().for_1 | 2173 | 1018.7 |
| CCNode.visit().if_3.for_1 | 8356 | 915.7 |
| CCSprite.draw() | 8594 | 766.9 |
| CCTexture2D.name() | 13085 | 537.5 |

### 4.2.1 Code optimisation

Both the correctness of software, and its energy-efficiency are primary design goals for app developers. In our case study our energy-aware programming approach is to focus on correctness first and then apply energy optimisations after correct working code is obtained. Clearly, a more comprehensive energy-aware development process would consider energy efficiency from the start, in choosing algorithms and data structures. However we show now that in our case study considerable optimisations are obtained from code that has been developed without energy-efficiency in mind.

The overview of energy-aware programming approach is firstly finding the most costly blocks, where we analyze the energy breakdown among the operations, and make changes to the code to remove, reduce or replace the costly operations.

We look into the top 10 costly blocks "In Application" (see Table 3). For example, *CCNode.visit()* is the entrance block of the *visit()* function; *CCNode.visit().if_4.for_1* is the body block of the `for` loop. These 10 blocks are distributed in seven methods, so the code review does not require heavy labor. We find four easy optimisation opportunities in blocks, such as *CCNode.visit()*, *CCNode.visit().if_4.for_1* and *CCTexture2D.name()*. There are also other opportunities in other blocks for saving energy, but requiring more efforts and gaining little. For example, *CCAtlas.updateValues().for_1* has several busy arithmetic expressions. Usually it is believed that replacing the busy expression with a variable could reduce energy cost, however in this case the overhead of variable declaration counteracts the energy-saving.

The four opportunities to improve the code are very simple and effective, but can only be discovered by the operation-level energy information. The changes will be shown as following.

**Program 1** Simplified parts of **original** code in *CCNode.visit()*

```
if (children_ != null) {
    if_body1;
}
draw(gl);
if (children_ != null) {
    if_body2;
}
```

**Program 2** The changed Program 1

```
if (children_ != null) {
    if_body1;
    draw(gl);
    if_body2;
} else {draw(gl);}
```

**If Combination.** This change is made in the most costly block *CCNode.visit()*, which has two comparisons, two Boolean operations, one *Method Invocation* and one parameter passing. In fact, the two `if` headers make the same comparison, as shown in Program 1. We change the code to Program 2, which combines the two `if` statements and meanwhile keep it logically consistent with Program 1. By these means each execution of the block can reduce one comparison, and when the condition is false, it can additionally reduce one *BlockGoto_if* .

**Program 3** Simplified parts of **original** code in *CCNode* class

```
public void visit(GL10 gl) {
        ......
    transform(gl);
        ......
}
public void transform(GL10 gl) {
    tranform_body;
}
```

**Program 4** The changed Program 3

```
public void visit(GL10 gl) {
        ......
    transform_body;
        ......
}
public void transform(GL10 gl) {
    transform_body;
}
```

**Inner-Class Method Inline.** When "In Application", the *transform()* function is invoked 18903 times and mostly by the *visit()* function. We change the Program 3 to Program 4 by inserting the body of *transform()* into *visit()*, meanwhile remaining the original *transform()* function in case that other parts of the code call it. This change can largely decrease the number of *transform()*'s *Method Invocation*s that are very expensive. However, it may be at the cost of losing readability of the code, which could also be compensated by adding explanatory comments.

**Loop-Invariant Code Motion.** *CCNode.visit().if_3.for_1* and *CCNode.visit().if_4.for_1* are entrance blocks of the two `for` loops as seen in Program 5. These two loops have a quantity, *children_.size()*, which is computed in each iteration but the value is constant. We thus hoist it outside the loop, as shown in Program 6, which can vastly save the energy of invoking and executing the *size()* function during every iteration. At the same time, we move the declaration of the *child* outside the loop, considering the cost of *Declaration_Object* is about 2.97 mJ and also in the top 30.

**Inter-Class Method Inline.** *CCTexture2D.name()* is the 10th most costly block and costs 537.5 J "In Application". However, its job is to simply get the value of the private member variable, *_name*, of the class *CCTexture2D*. This method has only two callers in the code. So we consider to make this variable public and let the two callers directly get access to the variable, which avoids the cost of *Method Invocation*. This change may harm the encapsulation of data, however, only one member of one class is changed. The trade-off between energy-saving and data encapsulation will be decided in the end by developers.

**Program 5** The full version of Program 2

```
if (children_ != null) {
for (int i=0; i<children_.size(); ++i) {
    CCNode child = children_.get(i);
    if (child.zOrder_ < 0) {
         child.visit(gl);
    } else
        break;
}
draw(gl);
for (int i=0; i<children_.size(); ++i) {
    CCNode child = children_.get(i);
    if (child.zOrder_ >= 0) {
        child.visit(gl);
    }
}
} else {draw(gl);}
```

**Program 6** The changed Program 5

```
CCNode child = new CCNode(); //added
int children_size = children_.size(); //added
if (children_ != null) {
    for (int i=0; i<children_size; ++i) { //changed
        child = children_.get(i); //changed
        if (child.zOrder_ < 0) {
            child.visit(gl);
        } else
             break;
    }
draw(gl);
for (int i=0; i<children_size; ++i) { //changed
    child = children_.get(i); //changed
    if (child.zOrder_ >= 0) {
        child.visit(gl);
    }
}
} else {draw(gl);}
```

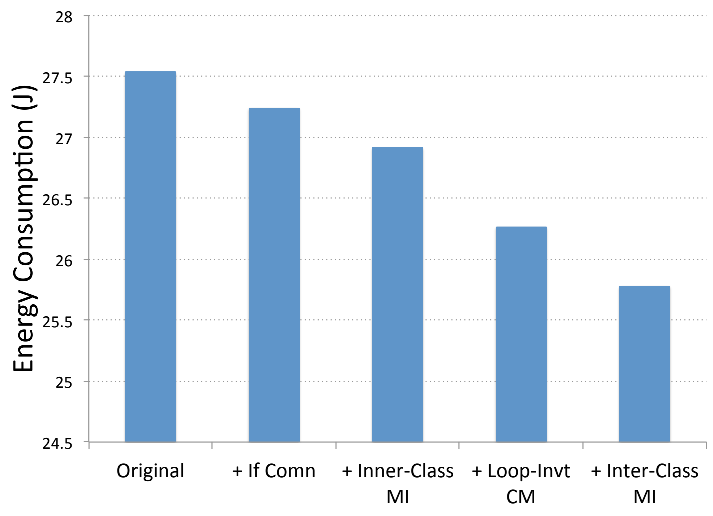


Figure 5: Energy consumption of the code without and with the changes in `Click & Move`.

### 4.2.2 Evaluation

Figure 5 illustrates the energy consumption of the software without and with the changes introduced in the previous section. From left to right, the bars indicate cumulative effects of the changes. For example, "+ *If Comn*" is the energy consumption of the code with "If Combination"; "+ *Inner-Class MI*" is the energy consumption of the code with the changes of both "If Combination" and "Inner-Class Method Inline". Totally, these four simple changes save 6.4% of the entire energy consumption without influencing the functionality of code. These changes are made in the basic part of the game engine, which most applications will use heavily, so any gain here can have fundamental impact. Furthermore, these changes are made with little knowledge about the algorithm of code, the developers who wrote the code are surely able to improve the code much more and achieve more energy-saving.

## 4.3 The `Orbit` Scenario

In this section, we briefly introduce the energy characteristics of `Orbit` scenario. Afterward, we improve the most costly blocks according to the types of expensive operations. In Section 4.3.3, we can see that the improvement can save as much as 50.2% of the overall energy consumption.

### 4.3.1 Energy Accounting

In the `Orbit` scenario, the block *CCGrid3d.blit().for_1* dominates the overall energy consumption. As shown in Figure 6, 80.9% of the entire cost is consumed by this block. The second

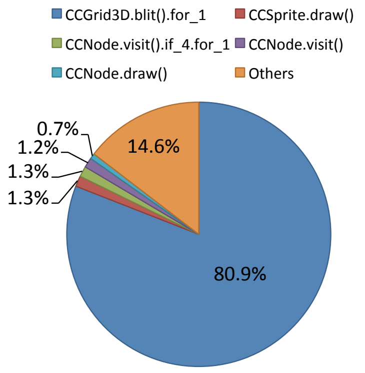


Figure 6: In `Orbit` scenario, the energy proportions of blocks "In Application".

costly block consumes only 1.3%. "In Application" here means running the `Orbit` scenario without removing any block. Later in Section 4.3.2, we will barely put attention on this single block, requiring fairly little effort to achieve improvements.

### 4.3.2 Code optimisation

Program 7 shows the original code of *CCGrid3D.blit().for_1*. In this block, the *Control Ops* (*BlockGoto_for* and *Field Reference*) use up 35.6% energy; *Boolean Ops* use up 20.5%; the assignments use up 16.7%; *Arithmetic Ops* use up 14.0%; *Lib Functions* use up 13.3%. We find three easy changes to reduce or replace the expensive operations.

**Loop-Invariant Code Motion.** In this block, the value of *vertices.limit()* is constantly 2112, we thus hoist it outside the loop and replace it with the variable *limit*, as shown in Program 8. This change avoids calls of *vertices.limit()* and at the same time decreases a small amount of *Field Reference*.

**Loop Unrolling.** Also as shown in Program 8, we duplicate the loop body eight times, which reduces the times of comparisons, *BlockGoto_for*s, assignments and additions. Note that, we set the value of increment as 24 since 24 is a factor of the *limit*, 2112.

**Full-Use of Library Function.** The job of Program 7 or Program 8 is getting all the elements in *vertices* one by one and putting them into *mVertexBuffer* one by one. The whole Program 7 in fact can be replaced by simply one line: *mVertexBuffer.put(vertices.asReadOnlyBuffer())*, which means putting the entire *vertices* into *mVertexBuffer*. This change realizes the same functionality using the already existing library function, which is one of the key library functions already compiled into native code.

**Program 7** The **original** code of *CCGrid3D.blit().for_1*

```
for (int i = 0; i < vertices.limit(); i=i+3) {
mVertexBuffer.put(vertices.get(i));
     mVertexBuffer.put(vertices.get(i+1));
     mVertexBuffer.put(vertices.get(i+2));
}
```

**Program 8** The changed Program 7

```
int limit = vertices.limit(); //added
for (int i = 0; i < limit; i=i+24) { //changed
     mVertexBuffer.put(vertices.get(i));
     mVertexBuffer.put(vertices.get(i+1));
     mVertexBuffer.put(vertices.get(i+2));
                    ...
                    ...
     mVertexBuffer.put(vertices.get(i+23));   //added
}
```

### 4.3.3 Evaluation

Figure 7 shows the cumulative effects of the code changes on energy consumption. Exceptionally, "*Full-Use LF*" does not take previous changes into account and means only replacing Program 7 with the built-in library function as stated above. We can see that loop-invariant code motion does not gain much energy saving because the *vertices.limit()* which is a library function as well uses a very small percentage of energy consumption. On the other hand, loop unrolling achieves 25.8% energy saving due to the reduction of amount of *Control Ops*, comparisons and assignments, which occupy most of the cost. And the most effective change is the replacement to a library function, saving 50.2% energy consumption because this library function has been complied into native code before execution, in contrast the java source code need run-time interpretation which is not free from energy consumption. The result indicates that it is a good

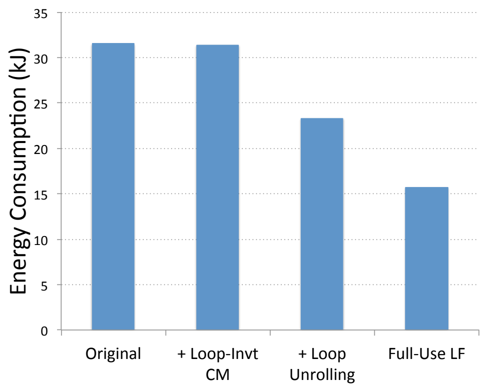


Figure 7: Energy consumption of the code without and with the changes in `Orbit`.

Table 4: In the `Waves` scenario, the top 10 costly blocks "In Application" and energy percentages of different kinds of operations in each block.

| Block ID | #Execs | Joules | Assi. | Decl. | Cont. | Func. | Bool. | Arit. | Libr. |
|---|---|---|---|---|---|---|---|---|---|
| CCGrid3D.blit().for_1 | 112193 | 8094.1 | 16.7% | 0% | 35.6% | 0% | 20.5% | 14.0% | 13.3% |
| CCVertex3D.CCVertex3D() | 40219 | 5232.0 | 27.2% | 0% | 10.0% | 62.8% | 0% | 0% | 0% |
| CCWaves3D.update().for_1.for_1 | 34604 | 4088.7 | 10.7% | 0% | 32.1% | 0% | 14.7% | 39.0% | 2.2% |
| ccGridSize.ccg() | 42275 | 3769.1 | 0% | 0% | 32.1% | 67.9% | 0% | 0% | 0% |
| CCGrid3DAction.setVertex() | 31856 | 3285.4 | 14.6% | 7.8% | 30.9% | 46.7% | 0% | 0% | 0% |
| CCGrid3DAction.originalVertex() | 36566 | 2891.3 | 19.1% | 10.2% | 40.3% | 30.4% | 0% | 0% | 0% |
| CCNode.getGrid() | 49119 | 2145.1 | 0% | 0% | 58.1% | 41.9% | 0% | 0% | 0% |
| ccGridSize.ccGridSize() | 10570 | 1173.8 | 30.3% | 0% | 31.6% | 38.0% | 0% | 0% | 0% |
| CCGrid3D.setVertex() | 3944 | 657.2 | 10.1% | 1.6% | 32.8% | 28.9% | 0% | 26.4% | 0.2% |
| CCGrid3D.originalVertex() | 2785 | 374.2 | 14.0% | 1.9% | 33.4% | 17.9% | 0% | 32.8% | 0% |

idea for developers to make a good use of library functions rather than implementing the same function themselves with java source code.

## 4.4 The `Waves` Scenario

In this section, similarly, we first analyze the energy characteristics of the blocks in the `Waves` scenario, based on which we modify the code and then evaluate the effects of changes on energy consumption.

### 4.4.1 Energy Accounting

Unlike the `Orbit` scenario where only one block dominates energy consumption, in `Waves` scenario, the costs of top seven blocks are at the same order of magnitude of kJ, as listed in Table 4. The *CCGrid3D.blit().for_1* is also employed in this scenario and is the most costly as well among all the blocks. The majority of blocks in Table 4 are directly or indirectly invoked by *CCWaves3D.update().for_1.for_1*, as shown in Program 9. And their jobs are mostly to set or get the values of member variables, so a large part of energy consumption goes to assignments and *Function Ops*. It was not expected that the code spends such a large amount of energy on simple setter and getter functions.

**Program 9** The **original** code in *CCWaves3D.update()*

```
int i, j;
for( i = 0; i < (gridSize.x+1); i++ ) {
  for( j = 0; j <(gridSize.y+1); j++ ) {
     CCVertex3D v = originalVertex(ccGridSize.ccg(i,j));
                          ...
     setVertex(ccGridSize.ccg(i,j), v);
  }
}
```

**Program 10** Program 9 after Method Inline & Code Motion

```
ccGridSize ccgridsize = new ccGridSize(0,0); //added
CCGrid3D ccgrid3d = (CCGrid3D) target.getGrid(); //added
CCVertex3D       v = new CCVertex3D(0,0,0); //added
int i, j;
for( i = 0; i < (gridSize.x+1); i++ ) {
    for( j = 0; j <(gridSize.y+1); j++ ) {
    ccgridsize.x=i;ccgridsize.y=j;  //added
    v = ccgrid3d.originalVertex(ccgridsize); //changed
                       ...
    ccgrid3d.setVertex(ccgridsize, v); //changed
    }
}
```

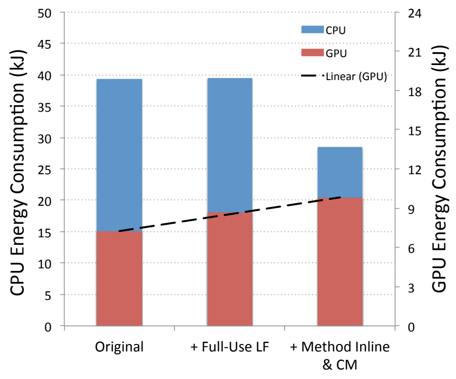


Figure 8: CPU and GPU Energy consumption of the code without and with the changes in `Waves`.

### 4.4.2 Code optimisation

**Full-Use of Library Function.** We have referred to the optimisation for *CCGrid3D.blit().for_1* in Section 4.3.2 where we replace the entire Program 7 with the one-line code, which makes use of library functions. We keep this change in this scenario. For other blocks, we come up with one modification as following.

**Method Inline & Code Motion.** As shown in Program 9, the three functions called in the inner loop body are *CCGrid3DAction.originalVertex()*, *ccGridSize.ccg()* and *CCGrid3DAction.setVertex()*, which respectively cost 2891.3 J, 3769.1 J and 3285.4 J “In Application”. Note that, *CC-Grid3DAction* is the parent class of *CCWaves3D*, so *originalVertext()* and *setVertex()* can be directly called without referring to their class names. As seen in Program 10, we unpack these three methods in this block: the first and fourth “added” lines are unpacked *ccGridSize.ccg()*; the second “added” and first “changed” lines are unpacked *CCGrid3DAction.originalVertex()*; the second “added” and second “changed” lines are unpacked *CCGrid3DAction.setVertex()*. This change removes all the *Method Invocation*s, parameter passing and value returns related to these three functions invoked by this block. Note that, the first three “added” lines are located outside the loop in order to reduce energy consumption of the process of initializing objects and calling *CCNode.getGrid()*.

### 4.4.3 Evaluation

Figure 8 shows the cumulative effects of changes on energy consumption of CPU and GPU (note that previous figures only showed the CPU energy consumption because the GPU energy consumption did not vary noticeably before), and the dashed line indicates the linear trend of the GPU energy consumption. In the case of game, usually the aimed frame rate is 60 Hz, when the game overloads CPU, the frame rate will decrease, and when the workload is light, even very light, the frame rate is generally fixed to 60Hz. The frame rate in "*Original*" is around 36Hz; that in "+ *Full-use LF*" is around 50Hz; that in "+ *Method Inline & CM*" is around 60Hz. The change of *Full-Use LF* (full use of library function) does not save energy for CPU since the execution of original `Waves` actually overloads the CPU capacity, so the improvement of code enhances the performance and enables the device to generate more frames every second. Consequently, the GPU does more work and consumes more energy, as seen in Figure 8. After this change, when we apply the method inline and code motion, 27.7% of the overall CPU energy is saved, and for the same reason GPU consumes slightly more. This experimental result shows that our approach not only saves energy but also potentially boosts performance, which benefits users doubly.

## 4.5 Conclusion

We proposed an energy-aware programming approach for mobile app development, which is guided by an operation-based source-code-level energy model (the construction of which is described in detail in the attachment). The general steps of the approach are as following: 1) we construct the operation-based energy model by mining the data generated in a range of well-designed execution cases; 2) based on the model, we capture the energy characteristics of the code; 3) we improve the code by removing, reducing or replacing the expensive operations in the costly blocks.

We evaluated this approach on a real-world game engine and on a physical Android development board with two ARM quad-core CPUs. The experimental result shows that our approach has a significantly positive impact on energy-saving. For different scenarios, this approach can save energy by from 6.4% to 50.2%. The result also indicates that the performance of code is a by-product as well of this approach, which potentially improves user experience more.

The optimisations were achieved by focussing on the mostly energy-consuming parts of the code, as identifed by an energy accounting process. The optimisations are manual and therefore it is vital to know that they are being applied where they can have the most effect.

The approach sometimes suggests a trade-off between the code structure and energy saving. In other words, some optimisations cause the structure of the code to be lost (for example breaking encapsulation). For example, in certain cases, we could unpack some thin methods that are highly-invoked in the code, at the cost of losing the integrity of the structure of the code to some extent. This choice is an important one which the energy-aware developer should understand, and depends to some degree on whether the code in question has to be modified and maintained.

# 5 Dynamic optimisations: energy-aware scheduling in multi-core environments

We have developed dynamic optimisation techniques for achieving energy efficient scheduling and allocation of tasks in multicore multithreaded environments with the possibility of voltage and frequency scaling. We have developed a framework for a multiobjective Evolutionary Algorithm (EA), where the objectives are energy and execution time, such that both may be subject to some constraints, given in terms of energy budgets or task deadlines [BLG13] (also as attachment D4.2.6 in this document). Based on this framework, different versions of the algorithm were implemented in order to address different types of problems:

1. *Deterministic algorithm.* The time and power of the tasks are expressed as deterministic values, which are obtained either by profiling or by static analysis at compile time. The energy of the whole schedule has been estimated either using a low level energy model of the targeted platform [BLG15a] (attachment D4.2.2), or by using static analysis [BLLG15a] (attachment D4.2.5). Different versions of this algorithm include:

   - Possibility to introduce *task migration and preemption*, i.e., the execution of a task can be stopped before finishing, and can be resumed afterwards on the same or different thread or core [BLG15a, BLLG15a, BLLG15b] (attachments D4.2.2, D4.2.5 and D4.2.3).
   - Possibility to *decrease accuracy in order to save energy* in applications that permit certain level of accuracy loss. Accuracy loss has been implemented through loop perforation, where for a previously defined (set of) loop(s), we skip every nth loop iteration [BLLG15b] (attachment D4.2.3).

2. *Stochastic scheduling*. We have proven that if the actual values of the power and execution time of the tasks are different from the estimations used to create the deterministic scheduler, the resulting scheduler does not have to be optimal in terms of energy and execution time. In this situation it is better to represent execution time and power as random variables with their corresponding distribution, and optimise the expected values of the energy and/or execution time of the scheduler. Two variations of this work have been implemented, based on the following assumptions:

   - The tasks are independent [BLG15c] (attachment D4.2.1).

- The tasks are dependent [BLG15b] (attachment D4.2.4). In this case the dependence has been modelled using copula theory, and it has been proven that more optimal results are achieved when modelling existing dependency.

3. *Modified YDS Algorithm* [BLLG15a] (attachment D4.2.5). YDS is a well known algorithm for task scheduling when the voltage and frequency can be scaled. Yet, it was invented in the 90s, and the hardware platforms have changed significantly since then. Multicore chips are practically a standard nowadays, while YDS was created for single core, and the static power forms a growingly important part of the total power consumption of a chip. To overcome these issues, we have performed two improvements: a) a method to stop decreasing voltage and/or frequency before the point the total power starts increasing again, and b) different methods to allocate tasks on different cores, which are then scheduled on each core using separate YDSs. In comparison with the EA algorithm, YDS is much faster, however, the final schedule provided by EA is much more optimal in terms of energy.

In all of the mentioned algorithms above it is very important to have an estimation of the resource consumption of a task, where the resource can be time or energy. For this purpose, we have greatly relied on the static analysis techniques implemented in the CiaoPP tool. Depending on the nature of the scheduling, i.e., if the deadlines or the energy budgets are hard (always have to be met), or soft (do not have be met all the time), we can use either average or worst case estimations. For the case of worst case estimations, which are both safe and accurate, we have combined evolutionary algorithms and static analysis.

Although the algorithms have been adapted for multicore multithreaded XMOS chips, they can be easily adapted to any multicore environment, including large scale data centers. Given that many tasks running in the datacenter at the same time are not related, the developed algorithms can be parallelized and efficiently executed using for example the *MapReduce* algorithm. On the other hand, experiments with static analysis have demonstrated that any a priori knowledge about resource consumption of a task, or a program, can provide better results in terms of additional energy savings: we have seen that the EA scheduler, which used both time and energy estimations, performs much better in terms of energy savings than the YDS algorithm, which uses only time estimations.

## References

[BLG13] Z. Banković and P. Lopez-Garcia. Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable XMOS Chips. In *Hybrid Artificial Intelligent Systems (HAIS 2013)*, volume 8073 of *Lecture Notes in Computer Science*, pages 401–410. Springer, 2013.

[BLG15a] Z. Banković and P. López-García. Energy Efficient Allocation and Scheduling for DVFS-enabled Multicore Environments using a Multiobjective Evolutionary Algorithm. In *Genetic and Evolutionary Computation Conference, GECCO 2015, Companion Material Proceedings*, pages 1353–1354. ACM, 2015.

[BLG15b] Z. Banković and P. López-García. Improved Energy-aware Stochastic Scheduling based on Evolutionary Algorithms via Copula-based Modeling of Task Dependences. In Álvaro Herrero, Javier Sedano, Bruno Baruque, Héctor Quintián, and Emilio Corchado, editors, *International Conference on Soft Computing Models in Industrial and Environmental Applications (SOCO 2015)*, volume 368 of *Advances in Intelligent Systems and Computing*, pages 153–163. Springer International Publishing, 2015.

[BLG15c] Zorana Banković and Pedro Lopez-Garcia. Stochastic vs. Deterministic Evolutionary Algorithm-based Allocation and Scheduling for XMOS Chips. *Neurocomputing*, 150:82–89, February 2015.

[BLLG15a] 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)*, volume 458 of *IFIP Advances in Information and Communication Technology*, pages 478–493. Springer International Publishing, 2015.

[BLLG15b] 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 Enrique Onieva, Igor Santos, Eneko Osaba, Héctor Quintián, and Emilio Corchado, editors, *Hybrid Artificial Intelligent Systems (HAIS 2015)*, volume 9121 of *Lecture Notes in Computer Science*, pages 690–701. Springer International Publishing, 2015.

[BR15] Denis Bogdanas and Grigore Roşu. K-java: A complete semantics of java. *SIGPLAN Not.*, 50(1):445–456, January 2015.

[LG15] Xueliang Li and John P. Gallagher. Fine-grained energy modeling for mobile application source code. Technical report, Roskilde University, December 2015. submitted for publication.

# Attachments

# Attachment D4.2.1

## Stochastic vs. Deterministic Evolutionary Algorithm-based Allocation and Scheduling for XMOS Chips

Published in Neurocomputing, Vol. 150, pages 82–89, Elsevier, February 2015.

Neurocomputing 150 (2015) 82–89

Contents lists available at ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

# Stochastic vs. deterministic evolutionary algorithm-based allocation and scheduling for XMOS chips

Zoran Banković[a,*], Pedro López-García[a,b]

[a] IMDEA Software Institute, Madrid, Spain
[b] Spanish Council for Scientific Research (CSIC), Spain

ARTICLE INFO

*Article history:*
Received 18 November 2013
Received in revised form
9 April 2014
Accepted 15 June 2014
Available online 2 October 2014

*Keywords:*
Deterministic and stochastic scheduling
Multiprocessor multithreaded architectures
Dynamic voltage and frequency scaling
Evolutionary algorithms
Multi-objective optimization

ABSTRACT

We present an approach based on multi-objective evolutionary algorithms for the automatic scheduling and allocation of tasks in a multiprocessor multithreaded architecture, together with an assignment of the appropriate voltage and frequency of each processor in a way the overall energy consumed by the execution of the tasks is optimized and all task deadlines are met. We have implemented both a deterministic scheduling algorithm, where the execution time and the energy consumption of different tasks have a known deterministic value, and a stochastic scheduling algorithm, where the execution time and energy are treated as random variables with corresponding probability density functions, given that in reality these values can vary significantly due to numerous reasons. It is assumed that execution time and energy consumption estimations, both for the deterministic and the stochastic case, are obtained by a static analysis process. It has already been proven for the case of makespan optimization that the stochastic scheduling is underestimated by its deterministic counterpart, and that in many real world situations, the stochastic scheduler outperforms the deterministic one. In this work we prove that for the tested scenario the stochastic scheduler for energy optimization outperforms its deterministic counterpart improving energy consumption by 15.4% in the best case.

## 1. Introduction

The most common approach for solving the problem of optimal task scheduling is to use a safe estimation of the values that the function to be optimized depends on, such as the execution time of each task or its power consumption. When such estimation gives a single (numeric) value for execution time (and power/energy), we refer to the problem as the *deterministic scheduling*. However, the execution time of a task in reality can vary considerably, due to a number of reasons, e.g. unknown memory access time, operating system effects that cannot be known in advance, etc. For this reason, it is more accurate to treat execution time, as well as energy consumption, which is closely related, as a random variable with a corresponding probability density and/or cumulative distribution function. We refer to this group of problems as *stochastic scheduling* problems. Moreover, there are state-of-the-art results of optimal scheduling for makespan optimization [1,2] that prove that in certain situations the deterministic scheduler provides results that significantly deviate from the optimal ones, and that better results can be obtained using stochastic scheduling. In this work we prove that this is also the case for energy consumption optimization.

Our objective is to optimize the energy consumption through scheduling and allocation of a set of tasks running on multi-processor/multicore[1] and multithreaded voltage and frequency scalable architectures designed by XMOS [3]. In such XMOS chips, threads are pipelined in four stages, where in each stage of the pipeline one instruction from a different thread is executed, so in essence we can say that the threads also run in parallel. Thus, we deal here with two levels of parallelism: multicore with multiple threads on each of them. Furthermore, the XMOS chips have the possibility to dynamically scale voltage and frequency, which can significantly contribute to energy consumption optimization, as we will explain in the following.

The dynamic power consumption due to the switching activity in digital CMOS circuits can be expressed with the following formula:

$$P = \alpha \cdot C_{eff} \cdot V^2 \cdot f$$

where $C_{eff}$ is the effective capacitance, $V$ the voltage supply, $f$ the operating frequency, and $\alpha$ the switching factor. If we decrease the

* Corresponding author.
*E-mail addresses:* zorana.bankovic@imdea.org (Z. Banković), pedro.lopez@imdea.org (P. López-García).

[1] These two terms are interchangeable throughout this paper.

http://dx.doi.org/10.1016/j.neucom.2014.06.077
0925-2312/© 2014 Elsevier B.V. All rights reserved.

voltage supply and the operating frequency, the dynamic power will decrease significantly. Moreover, the static power, which is the result of the leakage currents, also decreases quadratically with the voltage [4]. Thus, voltage and frequency decrease can achieve significant power and energy savings. This optimization technique is known as Dynamic Voltage and Frequency Scaling (DVFS). However, voltage and frequency decrease slows down the operation of the circuit, and has to be applied in a way the required timing deadlines of all the tasks are still met. Furthermore, the process of scaling voltage and/or frequency introduces additional latencies, which implies that we have to develop a set of requirements that define the applicability of this approach.

Regarding the application of DVFS to the XMOS chips, we assume that different processors/cores can have different ($V$, $f$) settings, while the threads running on the same processor/core at the same time must have the same ($V$, $f$) setting. We also assume that the wake-up time is big enough in relation to the execution times of the tasks to avoid the shutdown of separate processors. However, as we will see, our approach can be easily extended to take into account the possibility of shutdown.

Given a set of tasks and their corresponding deadlines, our objective is to provide a scheduling and allocation, and also assign voltage and frequency to each processor in a way the total energy consumption is optimized, while meeting the task deadlines. We assume that the tasks are heterogeneous, and they in general have different starting times and deadlines. We further assume that there is no precedence between tasks, and no preemption. Regarding the estimation of the corresponding values for execution time and power/energy consumption, in the deterministic case we assume that there are available analysis tools that give us such estimations, which are a necessary input to the scheduling algorithms we present here. Since this work has been done in the context of the ENTRA project [5], where tools for the estimation at compile time of the energy consumption and execution time of programs are being developed, we have direct access to such tools. For example, we can use the energy analyzer of the CiaoPP tool [6] already developed. In addition we could also use any existing timing analysis tools, for which great amount of work have been devoted. On the other hand, an ongoing work in the ENTRA project [5] is dedicated to deriving probability functions of both execution time and energy consumption, as well as the interdependence between different variables that represent time and energy of different tasks, which is the necessary input to the stochastic scheduler.

In general, the problem of scheduling and allocation is NP-hard. For this reason, different heuristic algorithms have been developed that are capable of obtaining sub-optimal solutions in real time, such as [7], which is based on Artificial Bee Colony for makespan minimization and machine occupation maximization, or [8], which compares the performances of a hybrid genetic algorithm, a hybrid simulated annealing and particle swarm optimization for the flow shop scheduling problem in a manufacturing supply chain. Many of these heuristic approaches use evolutionary algorithms (EAs), e.g. [9] for the vehicle routing problem, and in particular genetic algorithms (GAs) [10–12]. An EA is a well-known bio-inspired approach based on the principle of the survival of the fittest. Its most important advantage is a fast exploration of the search space, which allows the quick finding of acceptable solutions. This is the reason our scheduler is based on EAs. Since DVFS reduces energy, but increases execution time, these two magnitudes are clearly in conflict. For this reason, we use a multi-objective optimization approach, in order to find a trade-off between energy consumption and execution time. We also provide an appropriate representation of solutions that captures the two levels of parallelism, i.e., at both processor and thread level, and at the same time performs allocation and scheduling and identifies appropriate ($V$, $f$) settings in real time, exploring in this way the entire search space. As far as we know, the work presented in this paper is the first solution to mentioned type of problems.

The rest of the paper is organized as follows. Section 2 presents the most relevant related work and emphasizes the most important advantages of our approach. Section 3 details the sources of power consumption in CPUs and sets up the constraints that are the basis for generating a solution. Section 4 explains the problem that is being solved and points out the main differences between deterministic and stochastic scheduling. Section 5 details the implemented approach, while Section 6 explains the experimentation environment and presents the most significant results. Finally, Section 7 draws the most important conclusions and gives some directions for future work.

## 2. Related work

The related work to the one presented here falls into a great variety of topics, yet the ones with the closest relation are energy-aware scheduling approaches using DVFS, in particular the ones based on EAs (GAs). In the following we present these techniques and emphasize the main advantages of our approach.

### 2.1. Energy-aware deterministic scheduling

Since DVFS can provide significant energy savings, its optimal usage has been extensively studied. Some examples divide scheduling and allocation in two separate steps, such as the one given in [13], where in the first step the allocation problem is solved using Linear Programming, while in the second one the scheduling problem is solved for separate processors using Bin Packing. Another approach [10] solves the scheduling problem using a GA that integrates DVFS in the fitness function. However, such a division of the problem reduces the search space, since it becomes limited by the optimal solution of the first part of the problem, which does not always correspond to the global optimum. For this reason, we believe that better solutions can be achieved by solving the scheduling and allocation problem at the same time, while also accounting for the DVFS. There is one example of GA-based scheduling [11] that combines scheduling, allocation and power management in one process. However, it only deals with voltage scaling.

There is also a significant group of publications on using GAs for the optimal scheduling and allocation in multiprocessor systems with the DVFS feature. For example, the approach presented in [12] aims to minimize both energy and makespan as a bi-objective problem. The same problem is solved in another work [14], but using the island model of parallel GA populations. Another approach [15] treats 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 the solutions include the possibility of two levels of parallelism as in our work, where each processor can have a number of different threads executing in parallel.

### 2.2. Energy-aware stochastic scheduling

Stochastic scheduling has gained lots of interest over the years, since many different cases include uncertainty. In general, approaches to optimization under uncertainty include various modeling philosophies, the most important being the following ones:

- Expectation minimization.
- Minimization of deviations from goals.
- Minimization of maximum costs.
- Optimization over soft constraints.

Our approach clearly belongs to the first group. The solution presented in [16] is in the same group, however, it solves the stochastic scheduling problem by reducing it to the deterministic case. The benefit of this approach is the lower execution time, yet at the cost of decreased accuracy.

Regarding DV(F)S-based approaches, the typical ones are presented in [17,18]. The first one [17] belongs to the group that take advantage of soft constraints, in particular soft deadlines. However, it also divides the problem into two steps: in the first one it allocates cycles to tasks and schedules them to deliver performance guarantees, while in the second step it takes advantage of the soft deadlines to scale the voltage. The same approach is followed by the second GA-based work [18]. The only difference is that in this case the problem is divided into task mapping and task/voltage scheduling. As before, dividing the problem into two separate steps reduces the search space and does not always result in an optimal solution.

### 2.3. Advantages

Having presented the most important solutions from the state-of-the-art and their issues, we can emphasize the following advantages of our approach:

1. Thanks to the ENTRA project results, we can assume the availability of safe approximations of probability density functions for the execution time and power consumption, which give us the opportunity to gain higher accuracy when solving the scheduling problem.
2. By considering the scheduling, allocation, and voltage and frequency assignment at the same time, we explore the complete search space, and thus we have bigger chances of finding the most optimal solution.
3. Finally, our approach is the only one that supports two levels of parallelism: multicore with multiple threads running on each core.

## 3. CPU power consumption

The energy required to complete a (set of) task(s) in time $T$ on a processor, given its clock frequency $f$ and its voltage $V$, is defined by

$$E_{cpu,f,V} = \int_{t_0}^{t_0+T} P_{cpu,f,V}(t)\,\mathrm{d}t \tag{1}$$

where $P_{cpu,f,V}$ is the (time varying) power of the processor with the $(V, f)$ setting. This power can be expressed as

$$P_{cpu,f,V}(t) = P^{fix}_{cpu,V} + P_{idle,f,V} + P^{act}_{cpu,f,V}(t) \tag{2}$$

where $P^{fix}_{cpu,V}$ is the portion of the power that includes Phase Locked Loop (PLL) clock generator and leakage power consumption in the case of the XMOS chips [4], which is the part that only depends on voltage, not on frequency. $P_{idle,f,V}$ is the power spent when the processor is not executing any application, e.g., for maintaining the clock signal active. For a certain fixed $(V, f)$ setting, the sum of these two values does not change in time. For this reason, in the further text we will call this sum the *standing power* consumption, denoted $P^{std}_{cpu,V,f}$. This power can be easily obtained by measuring the CPU power when there are no running applications for each $(V, f)$ setting. $P^{act}_{cpu,f,V}(t)$ is the active power spent on switching activity during the execution of the application(s). Thus, formula (2) can be written as

$$P_{cpu,f,V}(t) = P^{std}_{cpu,f,V} + P^{act}_{cpu,f,V}(t) \tag{3}$$

which used in (1) gives the energy consumed during time $T$:

$$E_{cpu,f,V} = P^{std}_{cpu,f,V} \cdot T + \sum_{i=1}^{m} P_{i,f,V} \cdot T_i \tag{4}$$

where $P_{i,f,V}$ is the power spent by the application $i$, which is completed in time $T_i$, and $m$ is the number of threads (thus, the maximum number of applications that can be executed on one processor at any moment is $m$). In the case when the threads can complete more than one application within time $T$, formula (4) has the following form:

$$E_{cpu,f,V} = P^{std}_{cpu,f,V} \cdot T + \sum_{i=1}^{m} \sum_{j=1}^{k} P_{ij,f,V} \cdot T_{ij} \tag{5}$$

where $k$ is the maximum number of applications a thread can execute in time $T$, and $T_{ij}$ is the execution time of the application $j$ running on thread $i$.

### 3.1. Introducing dynamic voltage and frequency scaling

The main benefit of introducing DVFS lies in the significant energy reduction. However, we have to bear in mind that DVFS slows down the execution of the tasks, and in our scheduling problem, the task deadlines, if they exist, always have to be met. This gives us the first set of constraints, which will be explained in more detail in Section 3.1.1. In addition, we have to make sure that this process in reality decreases the energy consumption. Let us assume that $c_i$ is the number of clock cycles needed to complete the execution of task $i$ ($c_i = T_i \cdot f$, where $T_i$ is the execution time of task $i$ and $f$ is the clock frequency), $c_i^{max}$ is the maximum of all $c_i$, $\alpha_i$ is the switching activity when executing task $i$, $C_{eff}$ is the effective capacitance of the chip, and $k_1$ is the energy overhead of DVFS. Then, from the previous constraints we can conclude that the energy consumed at a clock frequency $f$ has the following form:

$$E(f) \approx k_1 + \frac{k_2}{f} + k_3 \cdot f^2 \tag{6}$$

where $k_2 \equiv P^{std}_{cpu,f,V} \cdot c_i^{max}$, and $k_3 \equiv C_{eff} \cdot \sum_{i=1}^{m} \alpha_i \cdot c_i$. Clearly, $E(f)$ has a minimum for $f = \sqrt[3]{k_2/2\,k_3}$, which means that a constant decrease in voltage and frequency does not necessarily mean a decrease in energy consumption. This gives us the second set of constraints that will be derived in Section 3.1.2. In the notation that we use in the following text we assume that variables are expressed using upper case letters and constants are expressed using lower case letters.

#### 3.1.1. Timing constraint

In the general case, for each new frequency $F_{new,i}$ of each processor $i$, after changing voltage and/or frequency, the deadlines still have to be met:

$$\forall i \in [1,n]\ \forall j \in [1,m] \left( T_{oh,i} + \frac{C_{ij}}{F_{new,i}} \le D_{ij} \right) \tag{7}$$

where $T_{oh,i}$ is the time overhead introduced by DVFS on processor $i$ and $C_{ij}$ is the number of clock cycles needed to execute the application $j$ on processor $i$ (and the corresponding execution time is $C_{ij}/F_{new,i}$, which is reasonable to assume, given that in the XMOS chips there is no cache memory and thus there are no pipeline stalls, nor cache misses that would introduce additional time overhead). Finally, $D_{ij}$ is the deadline of the task $j$ executed on processor $i$. We further have

$$\forall i \in [1,n] \left( T_{oh,i} = t_{oh_V} + T_{oh_{f,i}} \approx t_{oh_V} + \frac{10}{F_{old,i}} + \frac{2}{F_{new,i}} \right) \tag{8}$$

where $t_{oh_V}$ is the time overhead of performing voltage scaling (assumed constant) and $T_{oh_{f,i}}$ is the time overhead of performing frequency scaling on processor $i$, which takes 10 clock cycles at most of the old clock, and two cycles of the new clock [19]. If we want to include the possibility of shutting down the processor, the time overhead in this case would be the wake-up time. Finally, from (7) and (8) we get the timing constraints set:

$$\forall i \in [1,n]\ \forall j \in [1,m]\ (F_{new,i} \cdot (C_{ij}+2) \le D_{ij} - t_{oh_V} - 10/F_{old,i}) \tag{9}$$

where we assume that we know $t_{oh_V}$, and both $F_{old,i}$ and $F_{new,i}$ can take one value from the finite set of pre-established values $(V,f)$. Thus, the only unknown parameters are $C_{ij}$.

#### 3.1.2. Energy minimization constraint

The second set of requirements is derived from the condition of reducing the total energy during some known time $t$, high enough so that it permits the termination of all the applications. This implies the following condition:

$$\forall i \in [1,n], \quad \forall j \in [1,m]\ (t \ge \max_{i,j} D_{ij}) \tag{10}$$

Thus, for each processor, using formula (5) we have

$$\begin{aligned}\sum_{i=1}^{n} E_{old} &\ge \sum_{i=1}^{n} E_{new}\\ &\Rightarrow \sum_{i=1}^{n} p^{std}_{i,cpu,F_{old,i},V_{old,i}} \cdot t + \sum_{i=1}^{n}\sum_{j=1}^{m} p_{ij,V_{old,i},F_{old,i}} \cdot \frac{C_{ij}}{F_{old,i}}\\ &\ge \sum_{i=1}^{n} e_{oh} + \sum_{i=1}^{n} p^{std}_{i,cpu,F_{new,i},V_{new,i}} \cdot t + \sum_{i=1}^{n}\sum_{j=1}^{m} p_{ij,V_{new,i},F_{new,i}} \cdot \frac{C_{ij}}{F_{new,i}}\end{aligned} \tag{11}$$

where $e_{oh}$ is the energy spent on voltage and frequency scaling (assumed the same for all processors), $p_{ij,V_{old,i},F_{old,i}}$ and $p_{ij,V_{new,i},F_{new,i}}$ are the estimated total power consumptions of the application $j$ on processor $i$ in the different $(V,f)$ settings, while $p^{std}_{cpu}$ is the standing power explained in Section 3 in different settings. Finally, from (11), we get

$$\begin{aligned}&\sum_{i=1}^{n}\sum_{j=1}^{m}\left(\frac{p_{ij,V_{new,i},F_{new,i}}}{F_{new,i}} - \frac{p_{ij,V_{old,i},F_{old,i}}}{F_{old,i}}\right) \cdot C_{ij}\\ &\le t \cdot \sum_{i=1}^{n}\left(p^{std}_{i,cpu,F_{old,i},V_{old,i}} - p^{std}_{i,cpu,F_{new,i},V_{new,i}}\right) - n \cdot e_{oh}\end{aligned} \tag{12}$$

where the only unknown parameters are $C_{ij}$.

## 4. Problem definition: deterministic vs. stochastic

In essence, the objective of the scheduler that we propose in this paper is to optimize the total energy consumption. This is expressed by the following formula for a given set of $n$ heterogenous machines and a set of $k$ independent tasks, for a particular machine-task assignment $\chi$:

$$E_F(\chi) = \sum_{1 \le i \le n}\left(P_{st,i} \cdot T_F(\chi) + \sum_{1 \le j \le k} x_{i,j} \cdot p_{i,j} \cdot \tau_{i,j}\right) \tag{13}$$

where

$$T_F(\chi) = \max_{1 \le i \le n}\left\{\sum_{1 \le j \le k} x_{i,j} \cdot \tau_{i,j}\right\} \tag{14}$$

is the total execution time, $P_{st,i}$ is the standing power of the machine $i$, $x_{i,j}$ is a binary value, $x_{i,j} \in \{0,1\}$, that represents whether the task $j$ is executed on the machine $i$ ($x_{i,j}=1$) or not ($x_{i,j}=0$), $p_{i,j}$ is the (dynamic) power consumption of the task $j$ on the machine $i$, and $\tau_{i,j}$ is the execution time of the task $j$ on the machine $i$.

The deterministic scheduler uses the (deterministic) expected values of the execution time and power consumption: $E[\tau_{i,j}] = \overline{\tau_{i,j}}$, and $E[p_{i,j}] = \overline{p_{i,j}}$. Thus, in this case the scheduler considers the following value:

$$\widehat{E_F}(\chi) = \sum_{1 \le i \le n}\left(P_{st,i} \cdot \widehat{T_F} + \sum_{1 \le j \le k} x_{i,j} \cdot \overline{p_{i,j}} \cdot \overline{\tau_{i,j}}\right) \tag{15}$$

where

$$\widehat{T_F}(\chi) = \max_{1 \le i \le n}\left\{\sum_{1 \le j \le k} x_{i,j} \cdot \overline{\tau_{i,j}}\right\} \tag{16}$$

and, for the set of all possible schedules $\pi$, the objective is

$$\min_{\chi \in \pi}\{\widehat{E_F}(\chi)\} \tag{17}$$

and, if we are also interested in optimizing the execution time, we would add

$$\min_{\chi \in \pi}\{\widehat{T_F}(\chi)\} \tag{18}$$

However, the stochastic scheduler treats execution time and power consumption as random variables, and its objective is the following:

$$\min_{\chi \in \pi}\{\overline{E_F}(\chi)\} \tag{19}$$

i.e., minimizing the expected value of the energy consumed. Similarly, if we want to optimize the execution time, the objective is

$$\min_{\chi \in \pi}\{\overline{T_F}(\chi)\} \tag{20}$$

There exists a body of work which proves that $\overline{T_F}(\chi)$ is under-approximated by $\widehat{T_F}(\chi)$. Some examples are given in [1,2]. This means that if the actual execution times of the tasks are different from the expected ones used in the deterministic scheduler, the result may substantially deviate from the expected one. The authors of the cited works show that better schedules can be achieved by using the distributions of the task execution times.

The same can be proven for energy, using Jensen's inequality in the same way as used in [2]. Since $E_F(\chi)$ is a non-decreasing convex function, Jensen's inequality

$$E[f(X)] \ge f(E[X]) \tag{21}$$

can be applied on it. This gives us the following:

$$\begin{aligned}\overline{E_F} &= E\left[\sum_{1 \le i \le n}\left(P_{st,i} \cdot T_F + \sum_{1 \le j \le k} x_{i,j} \cdot p_{i,j} \cdot \tau_{i,j}\right)\right]\\ &= E\left[\sum_{1 \le i \le n} P_{st,i} \cdot \max_{1 \le l \le n}\sum_{1 \le j \le k} x_{l,j} \cdot \tau_{l,j}\right] + E\left[\sum_{1 \le i \le n}\sum_{1 \le j \le k} x_{i,j} \cdot p_{i,j} \cdot \tau_{i,j}\right]\\ &= \sum_{1 \le i \le n} P_{st,i} \cdot E\left[\max_{1 \le l \le n}\left(\sum_{1 \le j \le k} x_{l,j} \cdot \tau_{l,j}\right)\right]\\ &\quad + E\left[\sum_{1 \le i \le n}\sum_{1 \le j \le k} x_{i,j} \cdot p_{i,j} \cdot \tau_{i,j}\right]\\ &\ge \sum_{1 \le i \le n} P_{st,i} \cdot \max_{1 \le l \le n} E\left[\sum_{1 \le j \le k} x_{l,j} \cdot \tau_{l,j}\right] + \sum_{1 \le i \le n}\sum_{1 \le j \le k} x_{i,j} \cdot \overline{p_{i,j}} \cdot \overline{\tau_{i,j}}\\ &= \sum_{1 \le i \le n} P_{st,i} \cdot \max_{1 \le l \le n}\sum_{1 \le j \le k} x_{l,j} \cdot \overline{\tau_{l,j}} + \sum_{1 \le i \le n}\sum_{1 \le j \le k} x_{i,j} \cdot \overline{p_{i,j}} \cdot \overline{\tau_{i,j}}\\ &= \sum_{m_i \in M} P_{st,i} \cdot \widehat{T_F} + \sum_{m_i \in M}\sum_{v_i \in V} x_{i,j} \cdot \overline{p_{i,j}} \cdot \overline{\tau_{i,j}}\\ &= \widehat{E_F}\end{aligned} \tag{22}$$

Thus, $\overline{E_F}$ is under-approximated by $\widehat{E_F}$. In our experiments we will confirm that the stochastic scheduler in certain situations performs better than the deterministic one.

## 5. Our proposal for solving the scheduling problem

Both the deterministic and the stochastic approach for the optimal scheduling and allocation of tasks are based on the same multi-objective EA, the only difference is the objective function. The algorithm we propose is based on a simple EA. We have tested two common techniques for finding or approximating the Pareto-optimal front in multi-objective optimization problems, namely an improved version of Nondominated Sorting Genetic Algorithm (NSGA-II) [20] as well as an improved version of the Strength Pareto Evolutionary algorithm (SPEA2) [21], which we describe in the following. We also explain other important aspects of our implementation in more detail.

### 5.1. Multi-objective EA: NSGA-II vs. SPEA2

As a brief reminder, we will first explain the idea of Pareto-front and non-dominance. Without loss of generality, in minimization problems, a vector $x^{(1)}$ is partially less than another vector $x^{(2)}$ (denoted ($x^{(1)} \prec x^{(2)}$)), if no value in $x^{(2)}$ is less than its corresponding value in $x^{(1)}$, and at least one value in $x^{(2)}$ is strictly greater than its corresponding value in $x^{(1)}$. In this case we say that $x^{(1)}$ *dominates* $x^{(2)}$. Also, if none of the vectors dominate the other one, they are said to be *indifferent*. All the vectors that are not dominated by another are called *nondominated* and they form the so-called *Pareto front*. All the vectors belonging to the Pareto front can be a solution to the multi-objective problem.

As previously mentioned, the most common approaches for approximating the Pareto front with EAs are NSGA and SPEA. The original NSGA [22] employs ranking selection to emphasize good points and a niche method to maintain stable subpopulations of good points. A ranking is performed in the following way: nondominated solutions are assigned rank 1, rank 2 includes the solutions dominated only by those from rank 1, etc. The rank of each solution is actually its dummy fitness value. In order to get the fitness value, the solutions of the same rank are grouped in niches, for which is necessary to define a sharing value, which is the maximal of the possible distances between two solutions. The fitness value is then obtained by dividing the dummy fitness, i.e., the rank, by the size of the niche. However, the definition of the sharing value is not very clear, which is one of the reasons for criticism of NSGA. Other reasons for criticism are its computational complexity, $O(mN^3)$, and its non-elitism, i.e., there is no propagation of the solutions to the next generation. In order to overcome these problems the NSGA-II [20] was proposed. By performing a systematic book-keeping, the complexity was reduced to $O(mN^2)$, while the selection process is performed on a mating pool which consists of both parent and child populations, providing in this way the possibility of propagating solutions to the next generation. Finally, instead of defining a sharing value, it uses the *crowding distance*, which is the average distance between two points on either side of this point along each of the objectives.

On the other hand, SPEA [23] uses a regular population and an archive that is an external set containing the nondominated solutions found so far. The algorithm starts with an initial population and an empty archive, and performs the following steps. First, all nondominated population members are copied to the archive, and any dominated individuals or duplicates (regarding the objective values) are removed from the archive. If the size of the updated archive exceeds a predefined limit, some archive members are deleted by a clustering technique which preserves the characteristics of the nondominated front. Afterwards, fitness values are assigned to both archive and population members:

- Each individual $i$ in the archive is assigned a strength value $S(i) \in [0, 1)$ which is the number of population members $j$ that are dominated by or equal (i.e., indifferent) to $i$, divided by the population size plus one. For the nondominated solutions this also represents its fitness value $F(i)$.
- The fitness $F(j)$ of an individual $j$ in the population is calculated by summing the strength values $S(i)$ of all archive members $i$ that dominate or are equal to $j$, and adding 1 at the end.

The rest of the steps follow the standard EA.

The main criticism of SPEA concerns its fitness assignment, since all the individuals in the population dominated by the same elements from the archive are assigned the same fitness value, although some may dominate others. This means that if the archive contains only one solution, all the solutions from the population will have the same fitness value, in which case SPEA will behave as a random search algorithm. Moreover, if there are many indifferent solutions in a generation, none or little information can be obtained from the partial ordering defined by the dominance relation. In this case a density function is defined in order to guide the search process more effectively. The previously mentioned clustering technique can use this information for reducing the nondominated front while preserving its characteristics. However, it may lose some outer solutions which should be kept in the archive in order to maintain a good spread of the nondominated solutions.

In order to overcome these issues, SPEA2 [21] was proposed by the same authors. It uses a fine-grained definition of the fitness value, where the strength includes dominators from both the archive and the population. Furthermore, it is divided by a density value, which is a decreasing function of the distance to its $k$ nearest neighbors. In addition, the archive update is performed in a way its size remains constant, meaning that it can contain even dominated solutions, and it also prevents the removal of the boundary solutions.

There are many comparisons of NSGA-II and SPEA2 in different applications [24,25]. The general conclusion is that in the majority of the situations their performances are comparable, although SPEA2 outperforms NSGA-II as the number of objectives grows, while NSGA-II performs better for lower number of objectives. This suggests that in our case, NSGA-II might be a better solution, since we have only two objectives.

### 5.2. EA implementation

#### 5.2.1. Individual

The starting point, and one of the most important parts in designing an EA-based system is always the representation of a solution, i.e., an individual. In our case, the solution contains information about temporal and spatial allocation of each task. In other words, for each processor and each of its threads we should have an ordered (in time) set of tasks. However, since in this work we deal with DVFS, we have to add the information about the $(V, f)$ state of each processor. All the threads running on the same processor at the same time have the same $(V, f)$ setting, and we assume that different processors can have different $(V, f)$ settings, in order to solve the most general problem.

We assume that a solution to the scheduling problem is a permutation of the task identifiers (IDs), where their order also stands for the order of their temporal execution, assuming that each task has a unique ID. In order to solve the allocation problem, i.e., on which thread (and processor) each task is executed, we add delimiters to the permutation of the task IDs that define where the tasks are being executed, i.e., processor, thread and $(V, f)$ setting (the tasks between two delimiters are executed on the right-side one). In order to be able to distinguish the delimiters from the tasks, delimiters are coded as negative three-digit numbers, where the first digit stands for the processor, the second one for the thread on that processor, and the third one for the processor $(V, f)$ setting (recall that there is a finite number of settings). As an example, a part of a solution is depicted in Fig. 1, where tasks with

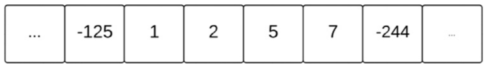


**Fig. 1.** An example of (part of) a solution (i.e., individual) representation.

IDs 1, 2, 5 and 7 are executed in that order on the thread 4 of the core 2, with the $(V, f)$ setting coded as 4. In the most general case, the order of delimiters is random. However, if two consecutive delimiters that belong to the same processor have different settings, this means that they are not being executed in parallel, since the state has to be changed. Representing individual in the described way has provided us with a relatively simple solution, which does not introduce great overhead when executing the EA.

### 5.2.2. *Population initialization*

Individuals in the initial population are created by randomly assigning tasks to random threads in random $(V, f)$ settings.

## 5.3. *Solution perturbations*

Given that all the tasks and all the delimiters are different, different solutions are always permutations of the set of tasks and the set of delimiters. This gives us the opportunity to use some of the permutation-based crossover operators, and in this case we are using partial match crossover, since it performed better in terms of the objective function than cycle crossover, and slightly better than order crossover in terms of the objective function and the execution time. Since the order of delimiters is not important in the most general case, this operator provides at the same time variety in consecutive changes of $(V, f)$ settings, as well as the capability of moving tasks from one thread to another. Regarding mutation, it is implemented in the way that two random threads exchange two random tasks with certain (low) probability.

### 5.3.1. *Objective functions*

The objective functions in the deterministic case are given by the already explained formulas (15) for energy consumption and (16) for makespan, and in the stochastic case by formula (19) for energy and (20) for makespan. The minimization of both values is required, while they are both penalized if the corresponding deadline is not met by multiplying the calculated value by 10. Since in this experiment we assume that random variables that define time and energy consumption of different tasks are independent (which is reasonable since they are random), the total energy or execution time of the tasks that are being executed sequentially on the same thread is a random variable obtained as the sum of the random variables that define energy or execution time of each task. Finally, in the case of the stochastic scheduling the expected values that we want to minimize are estimated using the Monte Carlo method.

## 6. Experimental evaluation

## 6.1. *Testing environment*

### 6.1.1. *XMOS chips*

Since this work is performed in the context of the ENTRA project, we target the architecture of the XMOS chips as a proof of concept. However, the same approach can be applied to any kind of DVFS-enabled multi-processor architectures, either with or without multiple threads. In the case of the XMOS chips, both voltage and frequency scaling are possible and both introduce a time overhead. All of the XMOS chips provide the possibility of frequency scaling due to the existence of a programable frequency divider. The time overhead introduced when changing the frequency is not more than 10 cycles of the old clock, plus two more cycles of the new clock, as expressed in Eq. (8).

However, only the XS1-SU01A-FB96 [26] chip provides the possibility of voltage scaling due to the existence of two DC–DC converters whose output voltage can belong to the range (0.6 V, 1.3 V). In order to apply DVSF, we should have a list of Voltage-Frequency $(V, f)$ pairs or ranges that provide correct chip functioning. The latency in this case is not controllable, and can be estimated in the following way. Since the switching frequency of the converter is 1 MHz, and assuming we have linear control, the bandwidth should be 1/10 to 1/7 of it, i.e., 150 kHz in the best case. Thus, the time it takes for the output voltage to stabilize is 1/150 kHz, which is around 6 μs.

Since this experiment is only a proof of concept, we assume that we have two different processors, where each processor has two different threads. We have experimentally concluded that the XMOS chips can function properly with the voltage and frequency levels given in Table 1. Column $P_{st}$ shows the typical static power consumption for both settings.

### 6.1.2. *Input data*

The input data to our scheduling algorithm consists of a set of independent tasks whose power consumption and execution time are given as random variables with a known probability density function. The following density functions are available at moment: Uniform, Constant, Exponential, Normal Chi-squared, Gamma, Pareto, Poisson, Binomial, Negative Binomial, and any combination of the previous expressed as a sum of products. A sampling from all the above density functions can be obtained by using a package implemented by Robert Davies [27]. In different $(V, f)$ settings, the power consumption is scaled with $V^2$ and $f$, and the corresponding execution time is scaled with $f$. Finally, the energy of a task is a random variable obtained as the product of their corresponding execution time and power random variables.

In order to validate the claim that the deterministic scheduler is not always optimal, we have created an optimal stochastic scheduler for a set of tasks with random power and execution time probability density functions taken from the above set. Then, we have created an optimal deterministic scheduler using the expected values of each distribution. Finally, in order to test the performance of both of them, we have created 10 different sets of tasks by taking a random sample of each underlying distribution.

## 6.2. *Obtained results and discussion*

We have conducted a set of experiments with the aim of evaluating the suitability of both the NSGA-II and the SPEA2 approaches for solving the scheduling problem. All the experiments have different input data, i.e., their task time and power distributions are different. In both the stochastic and the deterministic case the underlying EA has 100 individuals, it is evolved for 50 generations, the crossover probability is 90%, and the mutation probability is 40%. In the case of SPEA2 the archive contains 100 nondominated individuals.

In the first set of experiments we have evaluated the performance of SPEA2, taking as the final result the nondominated

**Table 1**
Valid $(V,f)$ levels and static power for XMOS chips.

| $V(V)$ | $f(MHz)$ | $P_{st}(mW)$ |
|---|---|---|
| 0.95 | 500 | 18.05 |
| 0.87 | 400 | 15.138 |

solution with the highest fitness value. The results of the experiments are presented in Table 2 and show the percentage of cases where the stochastic scheduler performs better than the deterministic one, as well as the maximal achieved improvement.

In the second set of experiments we have also evaluated SPEA2, but this time the final result is the nondominated solution with minimal energy, whose execution time is lower than the given deadline. The results are presented in Table 3.

In both sets of experiments it is evident that the number of cases improved by the stochastic scheduler is significant, as well as the amount of separate improvements. Furthermore, it is obvious that in our case, it is more beneficial to take the solution with the lowest energy than the nondominated solution with the highest fitness value, since in the second set of experiments we have obtained both higher energy savings and higher percentage of the solutions improved by the stochastic scheduler. Also, we can clearly see the trade-off between time and energy, i.e., bigger energy improvements lead to smaller execution time improvements and vice versa. However, the results obtained by SPEA2 are in general inconclusive, in the sense that it is not clear when the stochastic scheduler improves the results of the deterministic one. Bearing in mind that the good characteristics of SPEA2 become more relevant as the number of objectives grows, a possible explanation for their somewhat unsatisfactory results is that we only have two objectives. For this reason, we implemented the NSGA-II algorithm as well.

The last set of experiments is performed using the NSGA-II algorithm, taking the non-dominated individual with the minimal energy as the final solution. The results are presented in Table 4.

As we can observe, the results are much more satisfactory, since in almost all the experiments the stochastic scheduler improves the performance of its deterministic counterpart, achieving higher energy savings of up to 15.4%. In the case where the stochastic scheduler does not improve the energy consumption in all the experiments, it improves significantly the execution time. Thus, we can say that the results of the NSGA-II algorithm are quite satisfactory, and it represents a better choice for our problem than the SPEA2 algorithm.

**Table 2**
SPEA2 performance: the solution with the highest fitness value.

| ***Energy*** | | ***Time*** | |
|---|---|---|---|
| *improved* (%) | *max_improvement* (%) | *improved* (%) | *max_improvement* (%) |
| 40 | 1.6 | 40 | 29.7 |
| 20 | 4.2 | 0 | NA |
| 20 | 2.6 | 90 | 31.8 |
| 0 | NA | 70 | 19.6 |
| 90 | 8 | 0 | NA |
| 80 | 5.5 | 20 | 13.6 |
| 60 | 2.1 | 20 | 20.21 |

**Table 3**
SPEA2 performance: the solution with the lowest energy.

| ***Energy*** | | ***Time*** | |
|---|---|---|---|
| *improved* (%) | *max_improvement* (%) | *improved* (%) | *max_improvement* (%) |
| 30 | 5.5 | 0 | NA |
| 20 | 6.2 | 0 | NA |
| 40 | 3.1 | 70 | 20.1 |
| 70 | 1.5 | 10 | 5.1 |
| 20 | 2 | 40 | 25.1 |
| 100 | 13.5 | 10 | 24.4 |
| 50 | 4.2 | 0 | NA |

**Table 4**
NSGA-II performance: the solution with the lowest energy.

| ***Energy*** | | ***Time*** | |
|---|---|---|---|
| *improved* (%) | *max_improvement* (%) | *improved* (%) | *max_improvement* (%) |
| 100 | 13.2 | 0 | NA |
| 100 | 12.6 | 30 | 15.5 |
| 60 | 2.4 | 30 | 31.2 |
| 100 | 12.8 | 0 | NA |
| 100 | 14.8 | 0 | NA |
| 100 | 12.5 | 0 | NA |
| 100 | 15.4 | 0 | NA |

## 7. Conclusions and future work

In this work we have presented an approach for optimizing the energy consumption in a multiprocessor and multithreaded architecture. As far as we know, our approach is the only one that performs task scheduling and allocation, as well as appropriate voltage and frequency scaling at the same time in a single step and deals with two levels of parallelism. We have also experimentally shown that the stochastic scheduler can improve the results of the deterministic one in the situations where there is uncertainty in some of the parameters.

Our algorithms will be integrated in the tool set developed within the ENTRA project, and will take power and execution time estimations provided by the static analyzer of the tool set as input, given in terms of both deterministic values and probability density functions.

There are numerous possibilities to further improve the performance of our approach. For example, the XMOS chips have the possibility to automatically reduce the frequency of the processor if all of its threads are waiting for an event, and thus decrease the energy consumption even further. This feature will be included into future versions of our scheduler, as well as the possibility of shutting off separate components when they are not active.

Regarding the stochastic scheduler, one of its main drawbacks is its high execution time due to the application of the Monte Carlo method for estimating the expected value. Thus, we have to explore the possibility of using faster estimators. Another possibility for achieving speed-up is using parallel multi-objective optimization, such as the one presented in [28].

In addition, our current implementation assumes that both execution time and power consumption of different tasks are independent, which is reasonable in this case since their values are randomly chosen. However, in a real world situation this is not very realistic, having in mind that they are executed on the same platform. Thus, we have to study different possibilities of introducing the dependence between different variables. Finally, we plan to test the schedulers on real data.

## Acknowledgments

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 Madrid TIC-1465 *PROMETIDOS-CM* project.

## References

[1] A. Doğan, F. özgüner, Genetic algorithm based scheduling of meta-tasks with stochastic execution times in heterogeneous computing systems, Clust. Comput. 7 (2004) 177–190.

[2] T. Kidd, D. Hensgen, Why the mean is inadequate for accurate scheduling decisions, in: Proceedings of the 1999 International Symposium on Parallel

Architectures, Algorithms and Networks, ISPAN '99, IEEE Computer Society, Washington, DC, USA, 1999, pp. 262–267. URL: ⟨http://dl.acm.org/citation.cfm?id=850987.855693⟩.

[3] XMOS, xCore : Architecture Overview, Technical Report, XMOS Ltd., 2013. URL: ⟨https://www.xmos.com/en/download/public/xCORE-Architecture%28X9650D%29.pdf⟩.

[4] XMOS, Estimating Power Consumption for XS1-L Devices, Technical Report, XMOS Ltd., 2012. URL: ⟨http://www.xmos.com/download/public/Power-Consumption-For-XS1-L-Devices(X4271B).pdf⟩.

[5] Roskilde Univ., Univ. of Bristol, IMDEA Software Institute, XMOS, The ENTRA Project, ⟨http://entraproject.eu/⟩, 2013.

[6] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. Hermenegildo, K. Eder, Energy consumption analysis of programs based on XMOS ISA-level models, in: Pre-proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13), 2013.

[7] A. Madureira, I. Pereira, P. Pereira, A. Abraham, Negotiation mechanism for self-organized scheduling system with collective intelligence, Neurocomputing 132 (2014) 97–110, Innovations in Nature Inspired Optimization and Learning Methods Selected papers from the Third World Congress on Nature and Biologically Inspired Computing (NaBIC2011) Machines learning for Non-Linear Processing Selected papers from the 2011 International Conference on Non-Linear Speech Processing (NoLISP 2011).

[8] A. Noroozi, H. Mokhtari, I.N.K. Abadi, Research on computational intelligence algorithms with adaptive learning approach for scheduling problems with batch processing machines, Neurocomputing 101 (2013) 190–203.

[9] C. Chira, J. Sedano, J.R. Villar, M. Cámara, E. Corchado, Urban bicycles renting systems: modelling and optimization using nature-inspired search methods, Neurocomputing 135 (2014) 98–106, Advances in Learning Schemes for Function Approximation Selected papers from the 11th International Conference on Intelligent Systems Design and Applications (ISDA 2011).

[10] Y. Chang-tian, Y. Jiong, Energy-aware genetic algorithms for task scheduling in cloud computing, in: 7th ChinaGrid Annual Conference (CHINAGRID'12), 2012, pp. 43–48. http://dx.doi.org/10.1109/ChinaGrid.2012.15.

[11] V. Kianzad, S. Bhattacharyya, G. Qu, Casper: an integrated energy-driven approach for task graph scheduling on distributed embedded systems, in: 16th IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05), 2005, pp. 191–197. http://dx.doi.org/10.1109/ASAP.2005.23.

[12] M. Mezmaz, Y.C. Lee, N. Melab, E. Talbi, A. Zomaya, A bi-objective hybrid genetic algorithm to minimize energy consumption and makespan for precedence-constrained applications using dynamic voltage scaling, in: IEEE Congress on Evolutionary Computation (CEC'10), 2010, pp. 1–8. http://dx.doi.org/10.1109/CEC.2010.5586540.

[13] F. Paterna, A. Acquaviva, A. Caprara, F. Papariello, G. Desoli, L. Benini, An efficient on-line task allocation algorithm for qos and energy efficiency in multicore multimedia platforms, in: Design, Automation Test in Europe Conference Exhibition (DATE), 2011, March, pp. 1–6. http://dx.doi.org/10.1109/DATE.2011.5763025.

[14] M.-S. Mezmaz, Y. Kessaci, Y. Lee, N. Melab, E.-G. Talbi, A. Zomaya, D. Tuyttens, A parallel island-based hybrid genetic algorithm for precedence-constrained applications to minimize energy consumption and makespan, in: 11th IEEE/ACM International Conference on Grid Computing (GRID'10), 2010, pp. 274–281. http://dx.doi.org/10.1109/GRID.2010.5697985.

[15] P. Kumar, S. Palani, A dynamic voltage scaling with single power supply and varying speed factor for multiprocessor system using genetic algorithm, in: 2012 International Conference on Pattern Recognition, Informatics and Medical Engineering (PRIME), 2012, pp. 342–346. http://dx.doi.org/10.1109/ICPRIME.2012.6208369.

[16] C. Gong, X. Wang, W. Xu, A. Tajer, Distributed real-time energy scheduling in smart grid: stochastic model and fast optimization, IEEE Trans Smart Grid 4 (2013) 1476–1489.

[17] W. Yuan, K. Nahrstedt, Energy-efficient soft real-time cpu scheduling for mobile multimedia systems, in: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP'03, ACM, New York, NY, USA, 2003, pp. 149–163. http://dx.doi.acm.org/10.1145/945445.945460.

[18] B. Gorjiara, N. Bagherzadeh, P.H. Chou, Ultra-fast and efficient algorithm for energy optimization by gradient-based stochastic voltage and task scheduling, ACM Trans. Des. Autom. Electron. Syst. 12 (2007).

[19] XMOS, XS1-L Active Energy Conservation, Technical Report, XMOS Ltd., 2010. URL: ⟨https://www.xmos.com/download/public/XS1-L-Active-Energy-Conservation%28X7411A%29.pdf⟩.

[20] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast elitist multi-objective genetic algorithm: Nsga-ii, IEEE Trans. Evol. Comput. 6 (2000) 182–197.

[21] E. Zitzler, M. Laumanns, L. Thiele, SPEA2: Improving the Strength Pareto Evolutionary Algorithm, Technical Report, Computer Engineering and Networks Laboratory (TIK), 2001.

[22] N. Srinivas, K. Deb, Multiobjective optimization using nondominated sorting in genetic algorithms, Evol. Comput. 2 (1994) 221–248.

[23] E. Zitzler, L. Thiele, Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach, IEEE Trans. Evol. Comput. 3 (1999) 257–271.

[24] J. Lee, L. Choi, S. Park, Multi-objective genetic algorithms, nsga-ii and spea2, for document clustering, in: T.-h. Kim, H. Adeli, H.-k. Kim, H.-j. Kang, K. Kim, A. Kiumi, B.-H. Kang (Eds.), Software Engineering, Business Continuity, and Education, Communications in Computer and Information Science, vol. 257, Springer, Berlin, Heidelberg, 2011, pp. 219–227. http://dx.doi.org/10.1007/978-3-642-27207-3_22.

[25] T. Hiroyasu, S. Nakayama, M. Miki, Comparison study of spea2+, spea2, and nsga-ii in diesel engine emissions and fuel economy problem, in: IEEE Congress on Evolutionary Computation, CEC 2005, vol. 1, 2005, pp. 236–242. http://dx.doi.org/10.1109/CEC.2005.1554690.

[26] XMOS, XS1-SU01A-FB96 Datasheet, Technical Report, XMOS Ltd., 2012. URL: ⟨https://www.xmos.com/download/public/XS1-SU01A-FB96-Datasheet%28X7199A%29.pdf⟩.

[27] R. Davies, Random distributions, ⟨http://www.robertnz.net/⟩, 2013.

[28] M. Cámara, J. Ortega, F. de Toro, A single front genetic algorithm for parallel multi-objective optimization in dynamic environments, Neurocomputing 72 (2009) 3570–3579, Financial Engineering Computational and Ambient Intelligence (IWANN 2007).

**Zorana Banković** obtained her Electrical Engineer degree from the Faculty of Electrical Engineering at the University of Belgrade (Serbia) in 2005 and her Ph. D. degree from the Technical University of Madrid (UPM) in 2011. Currently she is a Post Doc at IMDEA Software Institute. Her research work has been dedicated to energy efficient security solutions for wireless sensor networks, anomaly detection and thermal-aware optimizations in data centers. Currently, her research work is mainly dedicated to energy aware software optimizations.

**Pedro López-García** received his Ph.D. degree in Computer Science from the Technical University of Madrid (UPM), Spain. He is currently Scientific Researcher at the Spanish Council for Scientific Research (CSIC) and Researcher at the IMDEA Institute for Research in Software Development Technologies. His main areas of interest include energy aware software engineering; abstract interpretation based program analysis and verification of nonfunctional program properties such as resource usage (user defined, energy, execution time, memory,etc.); parallel and distributed computing, and constraint logic programming.

# Attachment D4.2.2

## Energy Efficient Allocation and Scheduling for DVFS-enabled Multicore Environments using a Multiobjective Evolutionary Algorithm

**Published in Genetic and Evolutionary Computation Conference (GECCO 2015), pages 1353–1354, ACM, 2015**

# Energy Efficient Allocation and Scheduling for DVFS-enabled Multicore Environments using a Multiobjective Evolutionary Algorithm

Zorana Banković
IMDEA Software Institute
Campus Montegancedo s/n
28223 Pozuelo de Alarcon, Madrid, Spain
zorana.bankovic@imdea.org

Pedro López-García
IMDEA Software Institute and CSIC
Campus Montegancedo s/n
28223 Pozuelo de Alarcon, Madrid, Spain
pedro.lopez@imdea.org

## ABSTRACT

We present an approach for the automatic solving of the scheduling and allocation problem in multicore environments, as well as assigning optimal voltage and frequency levels to each core, using a multiobjective evolutionary algorithm (EA) where both energy consumption and the makespan are optimised and all deadlines are met. The main advantage of our approach is that we deal with all the aspects of the problem at once, which allows searching the whole solution space. In addition, the algorithm introduces the possibility of task migration, which is a novelty in EA-based approaches. Our results show that proper scheduling and allocation can provide significant energy savings in different scenarios: for our test case, and comparing to the well known YDS algorithm, up to 76% on average in the case of loose deadlines, and up 70% on average in the case of tight deadlines can be saved.

## CCS Concepts

•**Computing methodologies** → **Planning for deterministic actions;** •**Information systems** → *Information systems applications;*

## Keywords

Evolutionary algorithms, multiobjective optimization, scheduling, allocation, multicore

## 1. INTRODUCTION

The objective of this work is to efficiently solve the general problem of the scheduling and allocation of different tasks in multicore environments, and assign voltage and clock frequency to each core in a way the total energy consumption is minimised, while meeting all task deadlines. The tasks are characterised with their release time, execution time and deadline. In general, this problem is NP-hard, and it has been typically solved with heuristic algorithms since they can provide good solutions within an acceptable amount of time. For this reason, we use Evolutionary Algorithms (EAs). Furthermore, different levels of voltage and frequency are achieved by applying the Dynamic Voltage and Frequency Scaling (DVFS) technique. Since this technique reduces energy, but increases execution time, these two magnitudes are in conflict. For this reason, we use multiobjective optimisation.

All the existing solutions based on EA or Genetic Algorithms (GA) that treat a similar problem address it by dividing it into subproblems by first performing the scheduling, and then assigning the proper values of voltage and frequency. In this way the search space is reduced, which is not the case with our solution. Finally as far as we know, none of the existing solutions introduces the possibility of task migration.

We test our approach on multicore voltage and frequency scalable architectures designed by XMOS [2]. For this reason, our EA is based on an existing instruction-level energy model, which is described in [1]. However, the approach can easily be adapted to any multicore environment.

*GECCO '15 July 11-15, 2015, Madrid, Spain*
© 2015 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-3488-4/15/07.
DOI: http://dx.doi.org/10.1145/2739482.2764645

## 2. OUR PROPOSED APPROACH

We use the multiobjective NSGA-II algorithm to approximate the Pareto front, where the objectives are the energy consumption, calculated in the way presented in [1], and the execution time, calculated as the time spent until the last task finishes its execution. Both magnitudes should be minimised. However, all task deadlines have also to be met: the execution time objective is penalised by multiplying it by 10 for each task that does not met its deadline.

### 2.1 Representation of Individuals

Our proposed representation for individuals is shown in Fig.1. Each gene representing a unique task ID is followed by a gene representing the number of cycles of the task that is being executed in the current run. The order of task IDs represents the order of their temporal execution, i.e, scheduling. We also use negative two digit numbers to code the spatial allocation of the tasks in order to distinguish it from 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. The tasks following the allocation code are executed on that coded location.

The population is randomly initialised: tasks are assigned to random cores and random number of cycles are assigned

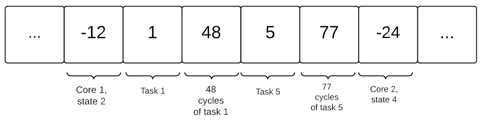


**Figure 1: An example of (part of) an individual.**

to tasks. However, the probability of selecting a core decreases as more tasks are assigned to it.

## 2.2 Solution Perturbations

### 2.2.1 *The Crossover Operator*

We have designed our own operator in the following way: each child takes the order of appearance of the tasks and their allocation from one of the parents. However, it can take the distribution of the number of cycles from any of the parents with equal probability.

### 2.2.2 *The Mutation Operator*

The mutation operator can perform different actions involving one or two tasks. Consider two tasks $i$ and $t$. In each generation we perform one of the following operations with the same probability: *swapping:* $i$ and $t$, together with their corresponding number of cycles, change their positions in the solution; *moving*: move $i$ to a random position $j$; and *changing the number of cycles:* assigns a different number of cycles to task $i$ in all of its appearances.

## 3. EXPERIMENTAL EVALUATION

The great majority of the existing work on applying DVFS concentrates only on dynamic power. However, it is not beneficial to scale down voltage and frequency indefinitely: as we keep decreasing the dynamic power, the static power is increased, which at some point becomes the predominant part, and as a result, the total power starts increasing again. This issue becomes important in the case when the tasks have loose deadlines. We have experimentally compared the energy savings obtained by our EA-based algorithm with the YDS algorithm [3], which was designed having in mind only dynamic power. The results, obtained by repeating the same experiment 20 times, are presented in Figures 2 and 3 for the scenarios where task deadlines are loose and tight respectively. As we can observe from Figure 2, energy savings are significant when task deadlines are loose: on average, up to 76.57% after 200 generations.

Moreover, Figure 3 shows that our EA-based algorithm can find a feasible solution even if task deadlines are tight, obtaining average savings of up to 70.4% after 150 generations. We believe that the reasons for such improvements are the following, in this order: taking into account the static power, the energy-aware scheduling of the tasks, and the good characteristic of EA of not getting stuck in a local optimum, which can happen in the YDS algorithm.

## 4. CONCLUSIONS AND FUTURE WORK

We have presented an approach for energy-aware scheduling, allocation and optimal DVFS assignment of a set of tasks in a multicore environment based on EAs. Our experimental results show the great potential of our approach. However, the energy model we use introduces significant

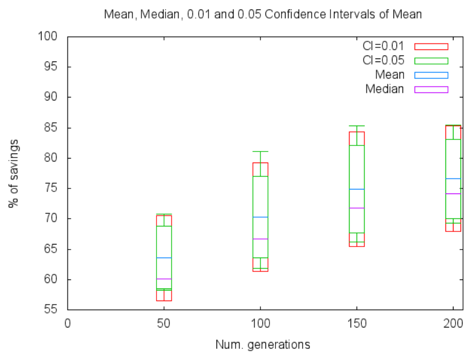


**Figure 2: Energy savings of our EA algorithm vs. YDS when task deadlines are loose.**

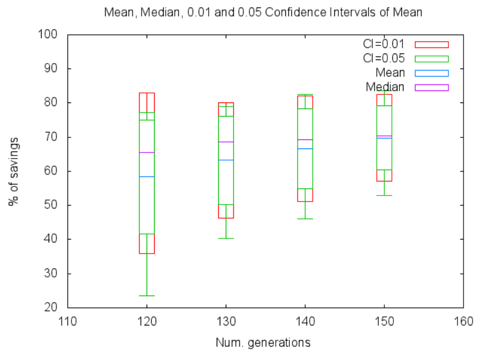


**Figure 3: Energy savings of our EA algorithm vs. YDS when task deadlines are tight.**

time overhead. In order to overcome this issue we plan to use a static analysis that estimates the energy consumed by concurrent tasks (without actually running them) which will significantly speed up our approach, since such estimations can be efficiently computed.

## 5. ACKNOWLEDGMENTS

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 Madrid TIC-1465 *PROMETIDOS-CM* project. We also thank XMOS Ltd. and Henk Muller in particular for providing benchmarks and useful information for our experimental evaluation.

## 6. REFERENCES

[1] 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.

[2] XMOS. xcore : Architecture overview. Technical report, XMOS Ltd., 2013.

[3] 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.

# Attachment D4.2.3

## Trading-off Accuracy vs. Energy in Multicore Processors via Evolutionary Algorithms Combining Loop Perforation and Static Analysis-based Scheduling

**Published in Hybrid Artificial Intelligent Systems (HAIS 2015), Lecture Notes in Computer Science, Vol. 9121, pages 690–701, Springer International Publishing, 2015**

# Trading-off Accuracy vs Energy in Multicore Processors via Evolutionary Algorithms Combining Loop Perforation and Static analysis-based Scheduling

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.** This work addresses the problem of energy efficient scheduling and allocation of tasks in multicore environments, where the tasks can permit certain loss in accuracy of either final or intermediate results, while still providing proper functionality. Loss in accuracy is usually obtained with techniques that decrease computational load, which can result in significant energy savings. To this end, in this work we use the loop perforation technique that transforms loops to execute a subset of their iterations, and integrate it in our existing optimisation tool for energy efficient scheduling in multicore environments based on evolutionary algorithms and static analysis for estimating energy consumption of different schedules. The approach is designed for multicore XMOS chips, but it can be adapted to any multicore environment with slight changes. The experiments conducted on a case study in different scenarios show that our new scheduler enhanced with loop perforation improves the previous one, achieving significant energy savings (31% on average) for acceptable levels of accuracy loss.

## 1 Introduction

Task scheduling and allocation for energy efficiency in multicore environments is a well-known $NP$-hard problem which can be efficiently solved with heuristic algorithms, such as evolutionary algorithms. One example is our approach for scheduling and allocation, which is based on evolutionary algorithms (EAs) [1]. The algorithm was shaped for its application to XMOS multicore chips, which give support for dynamic voltage and frequency scaling (DVFS) at chip level, i.e., all cores have the same voltage and frequency. However, the approach can be adapted to any multicore environment with slight modifications. In this work we want to deal with optimally scheduling tasks which can permit certain accuracy loss.

As a matter of fact, the great majority of today's processors are designed in a way that can provide a high level of accuracy. However, there are numerous applications that allow certain accuracy loss, which still permits them to function properly, such as video streaming, machine learning, etc. Since decreasing the accuracy is usually achieved by reducing the computational load, this can lead to both increase in performance and decrease in energy consumption, so here we deal with a trade-off between accuracy on one side and performance and/or energy on the other. One technique that achieves this is loop perforation [7], which in essence consists in skipping every $n$-th loop iteration, for a given $n$. Broadly speaking, accuracy can be considered as one aspect of quality of service (QoS), so we can say that in this work we deal with the QoS/energy trade-off.

Thus, in this work we solve the following scheduling problem: given a set of tasks with known release time and number of cycles to compute them, find proper allocation and scheduling of the tasks, as well as a $(V, f)$ assignment (i.e., voltage and frequency pair) to the cores in a way the total energy is minimised, while accuracy is maximised, meeting a minimal acceptable level of accuracy. Different levels of accuracy are achieved by applying the loop perforation technique with different $n$, where every $n$-th loop iteration is skipped.

Hence, we deal with two objectives: accuracy and energy. Accuracy is defined in terms of deviations of the output signal after applying the loop perforation, while in order to estimate energy consumption, we use an existing static analysis which, at compile time, with no need of executing the programs, and in a matter of seconds, gives a safe estimation of the energy consumed by programs. The energy consumption often depends on (the size of) input data, which is not known at compile time. For this reason, the static analysis provides the energy as a function of the input parameters, which is evaluated when input values are known at runtime. The energy consumption estimated by using the static analysis for a given scheduling is calculated as the sum of energies of the tasks running on different cores. This gives a safe upper bound on the total energy consumption, which is good enough for deciding which schedule consumes less energy, and can provide acceptable estimations of energy savings.

The rest of the paper is organised as follows. Section 2 gives more details of our proposed approach. Section 3 presents an experimental evaluation of it. Some related work is discussed in Section 4 and finally, some conclusions are drawn in Section 5.

## 2 Proposed Approach

### 2.1 Loop Perforation

The loop perforation technique consists in skipping some loop iterations, for example skipping every $n$-th iteration [7], where $n$ can be varied in order to trade accuracy with energy, i.e., for higher $n$, less instructions are skipped, so the accuracy is higher, while more energy is saved for lower values of $n$. This trade-off between accuracy and energy consumption justifies the usage of a multiobjective algorithm. As we will see in the following, in this work the loop perforation technique is implemented as one possibility for the mutation operator.

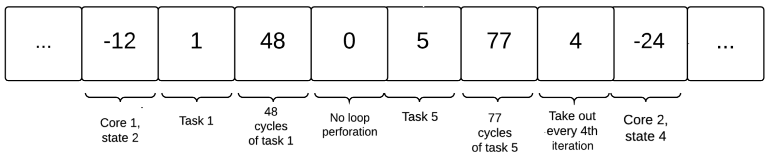


**Fig. 1.** Representation of an individual

### 2.2 Evolutionary Algorithm (EA)

The work presented in this paper is an extension of our previous work where we developed a custom algorithm based on an NSGA-II multiobjective evolutionary algorithm [1]. The conflicting objectives are accuracy and energy consumption, since we want to decrease the energy consumption, while maintaining the accuracy level as high as possible (always above a given threshold).

The non-dominated solutions are generated using the well-known NSGA-II algorithm [2], while the EA follows the standard steps of evolutionary algorithms: initialisation, evolution, where the selection process is implemented as standard tournament selection, and our custom-made crossover and mutation operators are applied. In the following we give more detail on the particular improvements carried out in this work.

**Individual.** A solution to the problem we are solving has to contain information about scheduling and allocation of each task, how many cycles of each task are executed in the current run (since we support task migration), and voltage and frequency levels of the core at each moment. In this work we add a new dimension to the problem, which is the possibility to decrease accuracy through loop perforation and thus it also has to be encoded in the individual. For this reason, we add one more field after each task, which encodes $n$, i.e., the iterations which can be skipped in one or more loops previously identified in each task. An example of a part of an individual is given in Figure 1, and can be read in the following way: on core 1 in state 2 we execute in this order,

– 48 cycles of task 1, without performing loop perforation on it, and
– 77 cycles of task 5, where we skip every 4th iteration in the loop previously defined.

**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. The number of cycles of a task executed in each run, as well as the loop iterations to be skipped are also randomly chosen.

**The Crossover Operator.** Our custom crossover operator is designed in the following way:

- Each child preserves the order of appearance of the tasks, as well as their allocation from one of the parents,
- But, can take the distribution of the number of cycles, as well as the number of loop iterations to be skipped of one of them with equal probability.

**The Mutation Operator.** The mutation operator can perform different operations involving one or two tasks (designated as $i$ and $t$ in the following text). In each generation we perform one of the following operations with the same probability:

- *Swapping:* $i$ and $t$, together with their corresponding number of cycles and loop iterations to be skipped, change 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 which are executed in parallel.
- *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 the same state as $i$'s original state.
- *Changing the cycle distribution:* Randomly change distribution of the cycles of task $i$ between its appearances on different cores.
- *Loop Perforation:* For a random task $i$, assign randomly the number of loop iterations to be skipped, update the total number of cycles, i.e., decrease the total number of cycles for the amount corresponding to the cycles of the skipped loops, and share them randomly between the existing appearances of the task $i$ in the solution.

These operators are depicted in Fig.2:

- *Swapping:* Tasks 1 and 2 are swapped between cores 1 and 2 while both in state 1.
- *Moving*: First part of task 1 (40 cycles) are moved to core 2 before task 2.
- *Changing the number of cycles:* Task 1 now executes 25 cycles on core 1 in state 1 and 45 cycles on core 2 and state 2.
- *Loop perforation:* Task 1, where loop perforation has not been performed, now skips every 20th task in the defined loop, which results in decreased number of cycles, i.e., it has 60 cycles, where the first 35 cycles are executed in the first appearance of the task 1, while the remaining 25 cycles are executed in its second appearance.

**Objective Functions: Energy Consumption.** This objective represents the total energy consumption of the given schedule, and it should be minimised. It is given with the following formula:

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

where $P_{st,i}$ is the static power of the 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, $x_{i,j} \in \{0, 1\}$,

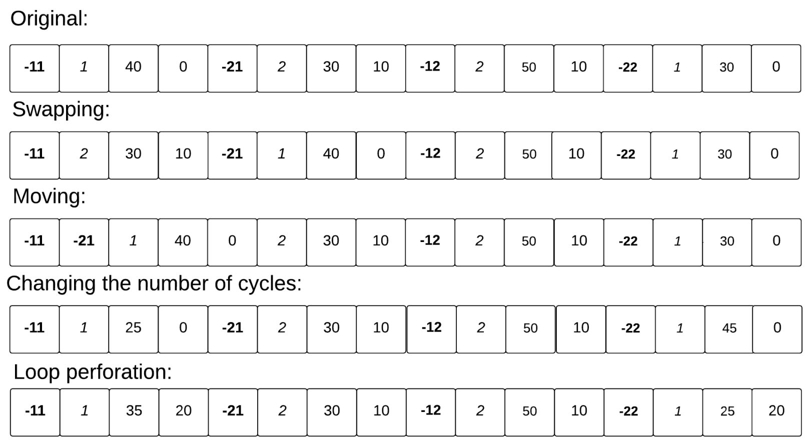


**Fig. 2.** Different possibilities for mutation

that represents whether the task $j$ is executed on the 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.** In this work accuracy is defined as an average error of the output after applying loop perforation, and it should be minimised. If a task performs some sort of signal processing, where the output is a digital signal consisting of a number of samples, the error is calculated as the Euclidean distance between the outputs obtained with and without loop perforation.

## 2.3 Energy Static Analysis as Input

In order to statically estimate the energy consumed by programs we use an existing static analysis. It is a specialization of the generic resource analysis presented in [8] for programs written in a high-level C-based programming language, XC [9], running on the XMOS XS1-L architecture, that uses the instruction-level energy cost models described in [3]. The analysis is general enough to be applied to other programming languages and architectures (see [4, 5] 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 [4, 5] 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 analyzer 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 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 ($N$): $fact_e(N) = 13\ N + 8$

## 3 Experimental Evaluation

### 3.1 Testing Environment

**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 multi-threaded, in this work we assume a single core architecture with 8 threads, which is the architecture for which we have an available energy model. All threads have their own register set and up to 4 instructions per thread can be buffered, which are scheduled in a way to minimize 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 $N = max(4, numberOfThreads)$.

DVFS is implemented at the chip level, which means that all the threads have the same voltage and frequency at the same time. All XMOS chips support frequency scaling. However, only the XS1-SU01A-FB96 [6] chip provides the possibility of voltage scaling enabled by two DC-DC converters whose output voltage belongs to the range (0.6V, 1.3V). In order to apply DVFS, we need 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.

**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 |

**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 equaliser implementation, which uses a cascade of Biquad filters. The energy consumed depends on the number of filters in the cascade, also known as banks `N`.

These filters are often used in signal processing, where some 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 number of coefficients: 85, 97, 109 and 121. Furthermore, we have used four implementations of the biquad program, with different number 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 2.3. This energy is expressed as a function of an input parameter $N$, which is known at run time only. In the case of FIR, $N$ is the number of coefficients, while in the case of the Biquad cascade, $N$ is the number of banks. These functions are given in Table 2. 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 this 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 2.2. 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.

## 3.2 Testing Scenario

We have tested our approach on a scenario of 32 tasks, where each task implements either an FIR or a Biquad cascade previously described. 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. Task

**Table 2.** Energy functions for 3 different pairs of voltage (V) / frequency (F, in MHz)

| | **V = 0.70** **F = 50** | **V = 0.75** **F = 100** | **V = 0.80** **F = 150** |
|---|---|---|---|
| $fir(N)$ | $74.93\ N + 124.5$ | $43.36\ N + 71.9$ | $33.41\ N + 55.2$ |
| $biquad(N)$ | $386\ N + 128$ | $223.6\ N + 74.2$ | $172.5\ N + 57.2$ |
| | **V = 0.80** **F = 300** | **V =0.87** **F = 400** | **V = 0.95** **F = 500** |
| $fir(N)$ | $20.14\ N + 33.2$ | $18.95\ N + 31.09$ | $19.15\ N + 31.3$ |
| $biquad(N)$ | $104.3\ N + 34.4$ | $98.31\ N + 32.4$ | $99.48\ N + 32.7$ |

deadlines do not exist. 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 static power consumption becomes more significant than dynamic power consumption. Thus, if we keep decreasing the dynamic power, the static power is increased at the same time, and as a result, the total energy consumption increases. The input signal to all tasks is a standardised set of input samples for testing in signal processing.

### 3.3 Obtained Results and Discussion

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

In order to illustrate the energy savings provided by loop perforation (referred to as *Case 1* in the following text), 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, the result of the training of both is a Pareto front of possible solutions with different trade-off between the objectives. Examples of Pareto fronts obtained in *Case 1* and *Case 2* are given in Figures 3 and 4 respectively. 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 3, with the following columns:

- *Column 1:* Maximal acceptable average error (or equivalently, minimal acceptable level of accuracy) of the final result.
- *Column 2:* Average energy of the final schedule obtained in a set of experiments of *Case 1* estimated by static analysis given in $mJ$ (mili Joules).
- *Column 3:* Average energy of the final schedule obtained in a set of experiments of *Case 2* estimated by static analysis given in $mJ$ (mili Joules).
- *Column 4:* Obtained savings expressed as % and calculated as $\frac{Column3-Column2}{Column3} \cdot 100$.

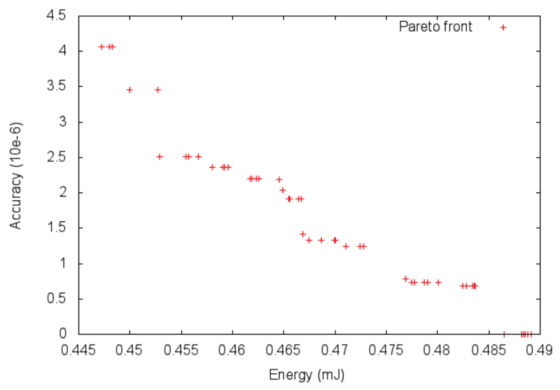


**Fig. 3.** Pareto front for Energy/Accuracy trade-off EA (Case1)

**Table 3.** Obtained savings 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 |

– *Column 5*: Statistics of the experiments expressed as 0.05 confidence interval, i.e., we can claim with 95% certainty that the final result will belong to this interval.

As we can observe, energy savings that can be obtained with loop perforation are significant and range from 3% to 40% in different experiments, even with small permitted level of error. As we increase the accepted level of average error, the savings increase, as expected, which is clearly depicted in Figure 5. However, the relationship between the accuracy and the 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 that lose acceptable accuracy by skipping less loop iterations (and hence achieve smaller energy savings).

Fluctuations in the final result in different experiments appear due to the imprecision of the static analysis, since currently it gives an upper bound, rather than a realistic estimation of energy consumption. This can explain the big confidence intervals. Since the acceptable level of error is small, we could observe that in the final result only tasks that perform FIR could skip a few iterations, while some of the tasks that perform biquad could skip one iteration at most, since the number of iterations is bigger in FIR than in the case of the biquad cascade. In Table 4 we present an example of a part of an output containing tasks

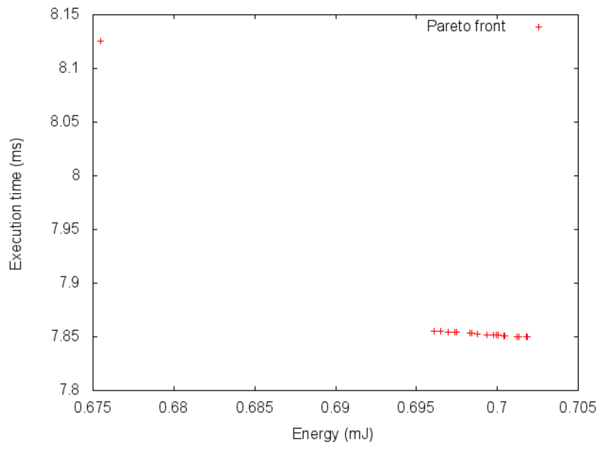


**Fig. 4.** Pareto front for Energy/Time trade-off EA (Case 2)

where loop perforation was applied, where the maximal acceptable error is $10^{-6}$. In the table, for each task, we show the original number of loop iterations, the number of loop iterations after applying loop perforation, and $N$, where every $N$-th loop is skipped. The actual error of this example is $7.8 \cdot 10^{-7}$, but we still achieve significant energy savings.

## 4 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., which has to be provided according to the signed Service Level Agreement (SLA) between the provider and the customer [11, 10, 12]. Multiobjective genetic algorithms were used in [12] to minimize cost and execution time, since they can be in conflict. A similar approach is presented in [11]. However, in the recent past, energy consumption has become a bottleneck, so it has become very important to reduce it. One such work is given in [10], where the authors try to minimize energy and maximize QoS at the same time in a Cloud Computing environment. The multiobjective optimisation problem is solved using particle swarm optimisation.

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 transforming the code, in our case by using loop perforation.

## 5 Conclusions

In this work we have presented an approach for energy efficient scheduling in multicore environments, adapted to multicore XMOS processors, where signifi-

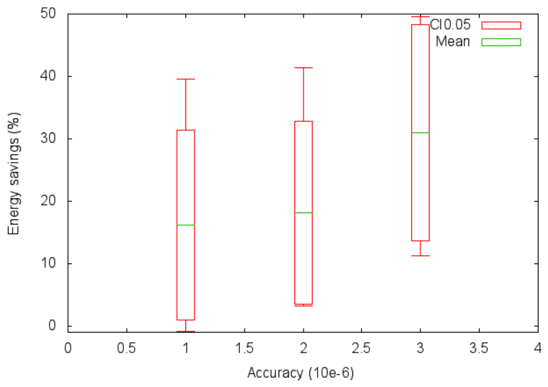


**Fig. 5.** Energy savings for different accuracy levels

cant additional energy can be saved if a certain level of accuracy reduction in final result is allowed. Accuracy reduction is performed by using the loop perforation technique. Our experimental results show that, even with small acceptable levels of error in the result, significant energy savings can be obtained.

However, the energy estimation of different schedules is based on a static analysis that can only provide an upper bound. Although it is still capable of providing energy savings, better results could be achieved with more precise energy estimations. For this reason, we are developing an energy analysis of concurrent program, which is expected to provide additional savings.

**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 Madrid TIC-1465 *PROMETIDOS-CM* project.

## References

1. Z. Banković and P. Lopez-Garcia. Stochastic vs. Deterministic Evolutionary Algorithm-based Allocation and Scheduling for XMOS Chips. *Neurocomputing*, pages 82–89, 2014.
2. 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.
3. S. Kerrison and K. Eder. Energy modelling of software for a hardware multi-threaded embedded microprocessor. *ACM Transactions on Embedded Computing Systems (TECS)*, 2015. To appear.

**Table 4.** Result of an experiment: tasks whose final number of loop iterations has been changed.

| Task | Original num. of loop iterations | Final num. of loop iterations | N |
|---|---|---|---|
| *FIR97-1* | 97 | 87 | 9 |
| *FIR85-1* | 85 | 76 | 9 |
| *FIR121-1* | 121 | 108 | 9 |
| *FIR109-1* | 109 | 104 | 21 |
| *FIR97-2* | 97 | 96 | 96 |
| *FIR85-2* | 85 | 84 | 84 |
| *FIR121-2* | 121 | 120 | 120 |
| *FIR109-2* | 109 | 108 | 108 |
| *FIR97-3* | 97 | 87 | 9 |
| *FIR85-3* | 85 | 76 | 9 |
| *FIR121-3* | 121 | 108 | 9 |
| *FIR109-3* | 109 | 97 | 9 |
| *FIR85-4* | 85 | 84 | 1 |
| *FIR121-3* | 121 | 81 | 3 |
| *FIR109-3* | 109 | 97 | 9 |

4. 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 the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'13)*, 2014.
5. P. López-García, editor. *Initial Energy Consumption Analysis.* ENTRA Project: Whole-Systems Energy Transparency (FET project 318337), April 2014. Deliverable 3.2, http://entraproject.eu.
6. XMos Ltd. Xs1-su01a-fb96 datasheet, november 2012.
7. Henry Hoffmann Sasa Misailovic, Stelios Sidiroglou and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proc. of FSE'11*. ACM Press, 2011.
8. A. Serrano, P. Lopez-Garcia, and M. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, 14(4-5):739–754, 2014.
9. D. Watt. *Programming XC on XMOS Devices.* XMOS Limited, 2009.
10. Sonia Yassa, Rachid Chelouah, and Rachid Chelouah andBertrand Granado. Multi-objective approach for energy-aware workflow scheduling in cloud computing environments. *The Scientific World Journal*, 2013. Article ID: 350934.
11. Guangchang Ye, Ruonan Rao, and Minglu Li. A multiobjective resources scheduling approach based on genetic algorithms in grid environment. In *Grid and Cooperative Computing Workshops, 2006. GCCW '06. Fifth International Conference on*, pages 504–509, Oct 2006.
12. Jia Yu, Michael Kirley, and Rajkumar Buyya. Multi-objective planning for workflow execution on grids. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, pages 10–17, Washington, DC, USA, 2007. IEEE Computer Society.

# Attachment D4.2.4

## Improved Energy-aware Stochastic Scheduling based on Evolutionary Algorithms via Copula-based Modeling of Task Dependences

**Published in International Conference on Soft Computing Models in Industrial and Environmental Applications (SOCO 2015), Advances in Intelligent Systems and Computing, Vol. 368, pages 153–163, Springer International Publishing, 2015**

# Improved Energy-aware Stochastic Scheduling based on Evolutionary Algorithms via Copula-based Modeling of Task Dependences

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

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

**Abstract.** In this work we apply the copula theory for modeling task dependence in a stochastic scheduling algorithm. Our previous work, as well as the majority of the existing related works, assume independence between the tasks involved, but this is not very realistic in many cases. In this paper we prove that, when task dependence exists, better results can be obtained when it is modeled. Our results show that the performance of the stochastic scheduler is significantly improved if we assume a certain level of task dependence: on average 18% of the energy consumption can be saved compared to the results of the deterministic scheduler, along with 81% of improved test cases, versus 2.44% average savings when task independence is assumed, along with 50% of improved test cases.

## 1 Introduction

The most common approach when dealing with a system which depends on a group of random variables is to assume that the variables are independent, mainly because the mathematical apparatus becomes too complex, or simply because it is not possible to mathematically describe the underlying dependence. However, this simplification often results in assuming an initial scenario which is very different from reality, which limits the usefulness of the final result. An example of this approach is stochastic scheduling, where the relevant characteristics of the tasks are represented as random variables with the corresponding distribution function, and, as far as we know, in the majority of existing works, the variables describing different tasks are considered to be independent, or modelled with normal distributions [6], which is often not realistic.

The main objective of our work is to minimize the energy consumption in multithreaded multicore Dynamic Voltage and Frequency Scaling (DVFS) enabled platforms through optimal task scheduling. In general, the problem of scheduling and allocation is NP-hard. Thus, the problem has been addressed with different heuristic algorithms that are capable of obtaining sub-optimal solutions in real time due to their fast exploration of the search space. For this reason, our scheduler is based on an Evolutionary Algorithm (EA). Since DVFS reduces energy, but increases execution time, these two magnitudes are clearly in conflict. For this reason, we use a multi-objective optimisation approach in order to find a trade-off between energy consumption and execution time. We also provide an appropriate representation of solutions that captures two levels of parallelism, processor (core) and thread level parallelism, and at the same time performs allocation and scheduling and identifies appropriate voltage and frequency settings (i.e., $(V, f)$ pairs), exploring in this way the entire search space.

This work is a part of a bigger tool for scheduling based on EAs [3], to which we want to add the possibility of modelling dependence between the tasks. For this reason, in this work we experiment with modeling dependence between the execution time and power of different tasks using copulas [12], in particular Archimedean copulas [11]. However, if we wanted a stand-alone implementation on copula-supported scheduling, a promising approach would be to use Estimation of Distribution Algorithms [8]. The main advantage of copulas when modeling dependence is the fact that they do not depend on the marginal distributions. In this work we have decided to study the applicability of Archimedean copulas for two important reasons: they have been extensively studied and the mathematical apparatus for their manipulation is quite mature, and they are known to properly model the existing tail dependency between two variables, i.e., the possibility of achieving extreme values at the same time, which can be expected in our case. In particular, in this work we experiment with Gumbel copulas [11] due to their proper modeling of positive right-tail dependence, e.g., if one task takes more time due to a prolonged memory access, it will lead to longer execution time of all the tasks that are related to it, as well as energy consumption, which is important when dealing with a time and/or energy budget. However, it does not model negative dependence, i.e., achieving small values at the same time, which is not important in our case since it does not affect the budgets mentioned above.

The rest of the paper is organised as follows. In Section 2 we give a short overview of the copula theory, necessary for understanding and reproducing the results of our work, while in Section 3 we detail our proposed approach. Section 4 presents and discusses the obtained results. In Section 5 we list the most important related work. Finally, some conclusions are drawn in Section 6.

## 2 Copulas for Modeling Dependency

In this section we give a short survey on copulas [12]. In essence, the copula theory gives us a mathematical framework for describing dependence between the variables irrespectively of their underlying distribution functions.

**Sklar's Theorem (1959)** Let $H$ be a *continuous* two-dimensional distribution function with marginal distribution functions $F$ and $G$. Then there exists a copula $C$ such that

$$H(x, y) = C(F(x), G(y)) \implies C(x, y) = H(F^{-1}(x), G^{-1}(y)) \tag{1}$$

Thus, for any two distribution functions $F$ and $G$ and copula $C$, the function $H$ is a two-dimensional distribution function with marginals $F$ and $G$.

**Archimedean Copulas.** An important group of copulas are the Archimedean copulas, where the dependence level depends on one parameter. They are defined in the following way: let $\phi$ be a continuous strictly decreasing function from $\mathbf{I}$ to $[0, \infty]$ such that $\phi(1) = 0$, and let $\phi^{[-1]}$ denote the pseudo-inverse of $\phi$:

$$\phi^{[-1]}(t) = \phi^{-1}(t) \text{ for } t \in [0, \phi(0)] \text{ and } \phi^{[-1]}(t) = 0 \text{ for } t \geq \phi(0) \tag{2}$$

Then, if $\phi$ is convex, the function

$$C(u, v) = \phi^{[-1]}(\phi(u) + \phi(v)) \tag{3}$$

is an *Archimedean* copula and $\phi$ is called its *generator function*. In the case of the Gumbel copula used in this work the generator function is the following:

$$\phi(t) = (-\ln t)^{\theta} \text{ where } \theta \in [1, \infty). \tag{4}$$

**Monte Carlo for Copula-based Models.** Since in our work we use Monte Carlo simulation to calculate the expected value of random variables, in the following we show how it is integrated in the copula model [10].

If we want to find the expected value of a function $g : \mathbb{R}^d \to \mathbb{R}$ applied to a random vector $(X_1, X_2, ..., X_d)$ whose cumulative distribution function (*cdf*) is $H$, the expected values we need are calculated in the following way:

$$\mathbb{E}[g(X_1, X_2, ..., X_d)] = \int_{\mathbb{R}^d} g(x_1, x_2, ..., x_d) \mathrm{d}H(x_1, x_2, ..., x_d) \tag{5}$$

If $H$ is given by its copula model expressed with formula 1, formula 5 can be written in the following way:

$$\mathbb{E}[g(X_1, X_2, ..., X_d)] = \int_{[0,1]^d} g(F_1^{-1}(u_1), ..., F_d^{-1}(u_d)) \mathrm{d}C(u_1, ..., u_d) \tag{6}$$

If copula $C$ and marginals $F_1, .., F_d$ are known or estimated, the expected value can be approximated using the following Monte Carlo algorithm:

1. Draw a sample $(U_1^k, U_2^k, ..., U_d^k) \sim C, k = (1, 2, .., n)$ of size $n$ from copula $C$.
2. Calculate a sample of $(X_1, X_2, ..., X_d)$ by applying the inverse cdf of marginal functions:

$$(X_1^k, X_2^k, ..., X_d^k) = (F_1^{-1}(U_1^k), F_2^{-1}(U_2^k), ..., F_d^{-1}(U_d^k)) \sim H(k = 1, ..., n)$$

3. Approximate $\mathbb{E}[g(X_1, X_2, ..., X_d)]$ with its empirical value:

$$\mathbb{E}[g(X_1, X_2, ..., X_d)] \approx \frac{1}{n} \sum_{k=1}^{n} g(X_1^k, X_2^k, ..., X_d^k)$$

**Archimedean Copula Simulation.** In order to draw a sample from Archimedean copulas (step 1 of the previous Monte Carlo algorithm), we follow the algorithm for Laplace transform Archimedean copulas, which are all the copulas whose generator function $\phi$ is a Laplace transform of some function $G$. The algorithm has been proven correct in [10] and consists on the following steps:

1. Generate a pseudorandom variable $V$ whose *cdf* is $G$.
   - For the Gumbel copula used in this work, $V$ is a stable distribution (class of probability distributions allowing skewness and heavy tails), $St(1/\theta, 1, \gamma, 0)$, with $\gamma = (cos(\pi/2/\theta))^{\theta}$ and $\hat{G} = exp(-t^{1/\theta})$.
2. Generate independent and identically distributed random variables $(X_1, X_2, ..., X_d)$.
3. Return $U_i = \hat{G}(-\frac{\ln X_i}{V}), i = 1, ..., d$.

## 3 Our Proposed Approach based on EAs

In the following we present the main aspects of our EA implementation. Our multiobjective EA is based on the NSGA-II implementation [5], since in our previous work [3] the best results were obtained when this technique was applied.

**Individuals.** An individual, as a representation of a solution to the problem, has to contain information about temporal and spatial allocation of each task. A solution to the scheduling, i.e., temporal aspect of the problem is a permutation of the task identifiers (IDs), where their order also stands for the order of their temporal execution, assuming that each task has a unique ID. In order to solve the allocation problem, i.e., on which thread (and core) each task is executed, we add delimiters to the permutation of the task IDs that define where the tasks are being executed, i.e., core, thread and $(V, f)$ setting (the tasks between two delimiters are executed on the right-side one). In order to be able to distinguish the delimiters from the tasks, delimiters are coded as negative three-digit numbers, where the first digit stands for the core, the second one for the thread on that core, and the third one for the core $(V, f)$ setting (assuming that there is a finite number of settings, which is realistic). As an example, a part of an individual is depicted in Fig. 1, where tasks with IDs 1, 2, 5 and 7 are executed in that order on thread 4 of core 2, with the $(V, f)$ setting coded as 4. In the most general case, the order of delimiters is random. However, if two consecutive delimiters that belong to the same core have different $(V, f)$ settings, this means that they are not being executed in parallel, since the voltage and/or frequency have to be changed. Representing individuals in the way described above has provided us with a relatively simple approach, which does not introduce great overhead when executing the EA.

**Population Initialisation.** Since our problem setting in this work is simple (two cores with two thread each), individuals in the initial population are created by randomly assigning tasks to random threads in random $(V, f)$ settings. However, in more complicated problem settings with more cores and threads

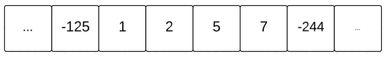


**Fig. 1.** An example of (part of) a solution (i.e., individual) representation.

per core, with task deadlines etc., we will have to consider adding a heuristic in order to provide a feasible solution in each run.

**Solution Perturbations.** Given that all the tasks and all the delimiters are different, different solutions are always permutations of the set of tasks and the set of delimiters. This gives us the opportunity to use some of the permutation-based crossover operators, and in this case we are using partial match crossover, since it performed better in terms of the objective function than cycle crossover, and slightly better than order crossover in terms of the objective function and the execution time. Since the order of delimiters is not important in the most general case, this operator provides at the same time variety in consecutive changes of $(V, f)$ settings, and the capability of moving tasks from one thread to another. Regarding mutation, it is implemented in a way that two random threads exchange two random tasks with certain (low) probability.

**Objective Functions.** The objective of the scheduler is to minimize the total energy consumption, as well as the execution time. Since in this work we apply DVFS, which decreases energy, but increases time, these two values are clearly in conflict, and thus the use of multiobjective optimisation is justified. In general, these values are expressed with the following formulas for a given set of $n$ heterogenous machines and a set of $k$ tasks, for a particular machine-task assignment $\chi$:

$$\begin{aligned} E(\chi) &= \sum_{1\le i\le n} (P_{st,i} \cdot T(\chi) + \sum_{1\le j\le k} x_{i,j} \cdot p_{i,j} \cdot \tau_{i,j}) \\ T(\chi) &= \max_{1\le i\le n} \{ \sum_{1\le j\le k} x_{i,j} \cdot \tau_{i,j} \} \end{aligned} \tag{7}$$

where $P_{st,i}$ is the static power of the machine $i$, $x_{i,j}$ is a binary value, $x_{i,j} \in \{0, 1\}$, that represents whether the task $j$ is executed on the machine $i$ ($x_{i,j} = 1$) or not ($x_{i,j} = 0$), $p_{i,j}$ is the (dynamic) power consumption of the task $j$ on the machine $i$, and $\tau_{i,j}$ is the execution time of the task $j$ on the machine $i$.

The objective of the stochastic scheduler is to minimize the expected value of these formulas:

$$\begin{aligned} &\min_{\chi\in\pi}\{\overline{E}(\chi)\} \\ &\min_{\chi\in\pi}\{\overline{T}(\chi)\} \end{aligned} \tag{8}$$

As in our previous work, these values are approximated using the Monte Carlo method, but here we have to introduce the necessary changes to account for the copula-based dependence:

- *Total Energy:* estimated in the Monte Carlo approximation presented in Section 2, taking $g(X_1, .., X_d) = \sum_{i=0}^{d} X_i$, where $d$ is the total number of tasks and $X_i$ are random variables representing the energy of each task.
- *Execution Time:* for each core approximated in the same way as the total energy, after that the maximum value between all the cores is taken.

In our implementation, we use the Mathematica system [1] to calculate the inverseCDF function (step 2 of the Monte Carlo method presented in Section 2) due to its capability to deal with all the probability functions used in this work. For this purpose, C++ executes Mathematica as an external program in MathLink mode [9].

## 4 Results and Discussion

### 4.1 Input Data

The input data to our scheduling algorithm consists of a set of tasks whose power consumption and execution time are given as random variables with a known probability density function. The following density functions are available at the moment: Uniform, Constant, Exponential, Normal Chi-squared, Gamma, Pareto, Poisson, Binomial, Negative Binomial, and any combination of the previous ones expressed as a sum of products. A sampling from all the above density functions can be obtained by using a package implemented by Robert Davies [4]. In different $(V, f)$ settings, the power consumption is scaled with $V^2$ and $f$, and the corresponding execution time is scaled with $f$. Finally, the energy of a task is a random variable obtained as the product of their corresponding execution time and power random variables.

### 4.2 Obtained Results and Discussion

In this work we experiment with controlled dependency in synthetic data. Dependency is introduced in a way that some of the random variables which describe execution time and power of the tasks have the same fixed distribution. As we have previously mentioned, dependence in Archimedean copulas is controlled with the $\theta$ parameter, which in the case of the Gumbel copula belongs to the interval $[1, \infty)$, where $\theta = 1$ stands for independence, while $\theta \to \infty$ stands for comonotonicity, i.e., maximal positive dependence. However, $\theta \geq 10$ is already considered as a significant level of dependence.

In order to check the possibility of improving the results of the stochastic scheduling by introducing copulas for modeling dependence, we have created an experiment where we fix the testing scenario, start with independence assumption, i.e., $\theta = 1$, and then increase the $\theta$ parameter in order to increase the level of dependence. Since the main claim of our work is that the stochastic scheduling may improve its deterministic counterpart, we check how the results are being improved as the level of dependence increases. In particular, we repeat the simulation for three different levels of dependence: $\theta = 1$, which is equivalent to independence, $\theta = 5$ and $\theta = 10$, which assumes a high level of dependence.

**Table 1.** Summary of average improvements.

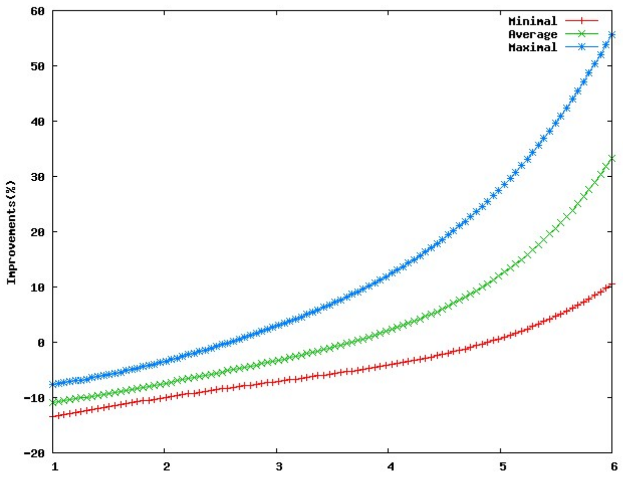
| **Test case ($\theta$)** | **Avg. % of improved test cases** | **Avg. improv. (%)** | **Avg. min. improv. (%)** | **Avg. max. improv. (%)** |
|---|---|---|---|---|
| 1 *(1)* | 50 | 2.46 | $-4.58$ | 12.1 |
| 2 *(5)* | 81.66 | 18 | $-3.51$ | 31.96 |
| 3 *(10)* | 68.75 | 8.42 | $-3.95$ | 22.75 |

After performing the training and obtaining the Pareto front, in all the cases we took the solution with minimal energy consumption from all the solutions belonging to the Pareto front. The testing was performed on different sets of test cases, each having 10 test cases generated randomly. The results are summarised in Table 1, where we can observe (from left to right) average % of test cases where the stochastic scheduler improves the deterministic one, along with average improvement, and average minimal and maximal improvement in all test cases. Figure 2 illustrates the evolution of different test cases (1-6 are different test case sets, each having 10 different test cases), sorted by their improvement. Note that negative numbers actually mean that the stochastic scheduler worsened the performances.

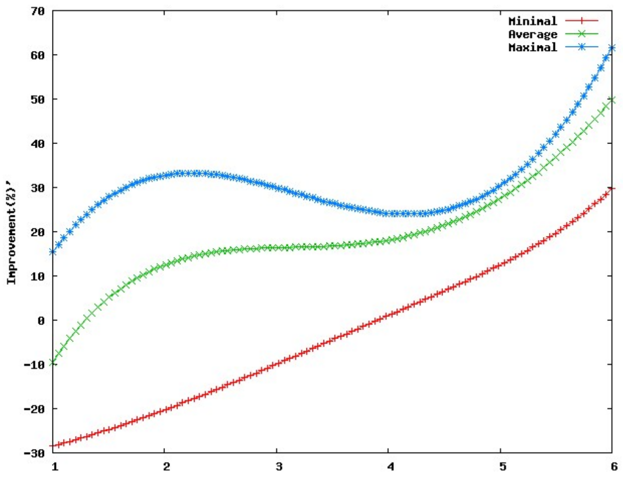
In the current implementation we rely on Mathematica to calculate the inverse cdf, which significantly increases simulation time, since it takes 6–8 hours on a 2.5GHz Intel Core i5 with 4GB DDR3.

From the table and the figures we can draw a few interesting conclusions:

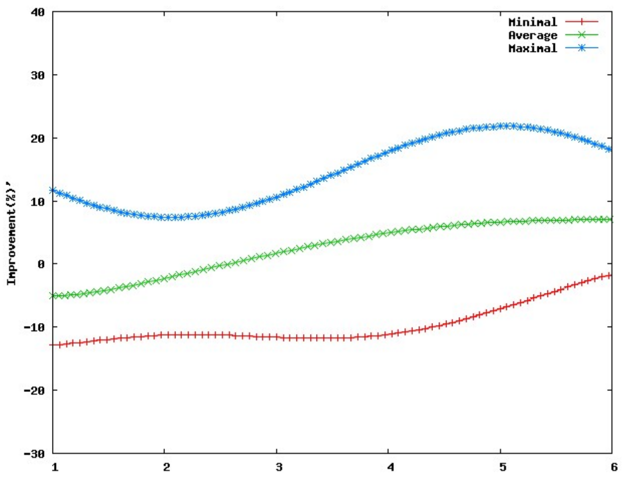
- The stochastic scheduler improves a significant number of test cases, even when assuming independence ($\theta = 1$).
- The best obtained results correspond to $\theta = 5$, which assumes certain level of dependence, although less than maximal positive dependence, which is close to $\theta = 10$. This confirm our expectation, since in our test cases some of the variables describing power and/or time are correlated, but not all of them.
- The maximal observed energy improvement achieved in a particular case is around 62%, while the maximal observed performance (in terms of energy savings) decrease is 28%, both belonging to the case $\theta = 5$ (Figure 2).
- The test cases that obtained better performances with the deterministic scheduler, i.e., whose performances were decreased after applying the stochastic scheduler (and which can be observed as "negative" improvement in both Table 1 and Figure 2), had values which were close to the average values of the corresponding distributions, which were used to design the deterministic scheduler. This behaviour was also expected, since the main idea of the stochastic scheduler is to improve the scheduling when the real data deviate significantly from the expected ones used to create the deterministic scheduler. However, from this testing scenario we were not able to properly decide the threshold level which would tell us when it is beneficial to start using the stochastic scheduler.



(a) $\theta = 1$



(b) $\theta = 5$



(c) $\theta = 10$

**Fig. 2.** Evolution of Minimal, Average and Maximal Improvements (%) for $\theta = 1, 5, 10$

## 5 Related Work

Stochastic scheduling has gained lots of interest over the years, since many different cases include uncertainty. In general, approaches to optimisation under uncertainty include various modeling philosophies, the most important being the following ones:

- Expectation minimisation.
- Minimisation of deviations from goals.
- Minimisation of maximum costs.
- Optimisation over soft constraints.

Our approach clearly belongs to the first group. The solution presented in [7] is in the same group, however, it solves the stochastic scheduling problem by reducing it to the deterministic case. The benefit of this approach is the lower execution time, yet at the cost of decreased accuracy.

Copulas have been used in different versions of scheduling-related problems. For example, in [2] they are used in power supply system scheduling to model the presence of uncertain renewables, such as wind and solar energy. Another work given in [6] assesses the schedule reliability of airport schedules using copulas. One more example is given in [13], where the authors use copulas to assess the schedule risk of a software development project. However, all of them use Gaussian or Student-t copulas, which can be applied only if the marginal distributions are either normal or Student-t, while in our work there is no restriction on the marginal distributions. As far as we know, there were no attempts to use copulas in the way presented in this work.

## 6 Conclusions

In this work we have studied the possibility of introducing dependence of both power consumption and execution time of the tasks which run on the same platform in order to improve the results of optimal task scheduling. Power consumption and execution time of the tasks are represented as random variables with known distribution functions, while their dependence is modeled with Gumbel copulas, whose $\theta$ parameter is varied in order to simulate different levels of dependence. As far as we know, this is the first work that uses copula-based dependence in the context of stochastic scheduling where the important aspects of the tasks are modeled using random distribution functions. If there is dependence present in the underlying data, our results show that the performance of the stochastic scheduler is significantly improved after assuming certain level of dependence: on average 18% of energy consumption can be saved compared to the results of the deterministic scheduler, along with 81% of improved test cases, versus 2.44% average savings when task independence is assumed, along with 50% of improved test cases. The obtained rates demonstrate the potential of the approach in improving results of stochastic scheduling when dependence between the tasks is present. In the future we plan to devote additional effort to providing faster simulation time, which would enable a real world implementation of the approach. Furthermore, we will test the approach on real world data.

**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 Madrid TIC-1465 *PROMETIDOS-CM* project.

## References

1. Mathematica. `http://www.wolfram.com/mathematica/`.
2. Sajjad Abedi, Gholam Hossein Riahy, Seyed Hossein Hosseinian, and Mehdi Farhadkhani. Improved stochastic modeling: An essential tool for power system scheduling in the presence of uncertain renewables. 2013.
3. Z. Banković and P. Lopez-Garcia. Stochastic vs. Deterministic Evolutionary Algorithm-based Allocation and Scheduling for XMOS Chips. *Neurocomputing*, pages 82–89, 2014.
4. Robert Davies. Random distributions. `http://www.robertnz.net/`, 2013.
5. 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.
6. Tony Diana. Improving schedule reliability based on copulas: An application to five of the most congested {US} airports. *Journal of Air Transport Management*, 17(5):284 – 287, 2011.
7. Chen Gong, Xiaodong Wang, Weiqiang Xu, and A. Tajer. Distributed real-time energy scheduling in smart grid: Stochastic model and fast optimization. *IEEE Transactions on Smart Grid*, 4(3):1476–1489, 2013.
8. Yasser Gonzalez-Fernandez and Marta Soto. copulaedas: An r package for estimation of distribution algorithms based on copulas. *Journal of Statistical Software*, 58(9):1–34, 2014.
9. Mathematica. *MathLink Reference Guide*, 1993.
10. A.J. McNeil, R. Frey, and P. Embrechts. *Quantitative Risk Management: Concepts, Techniques, and Tools*. Princeton Series in Finance. Princeton University Press, 2010.
11. Alexander J. McNeil and et al. Multivariate archimedean copulas, $d$-monotone functions and $l_1$-norm symmetric distributions, 2009.
12. Roger B. Nelsen. Properties and applications of copulas: A brief survey. In *First Brazilian Conference on Statistical Modelling in Insurance and Finance*, pages 10–28, 2003.
13. Dengsheng Wu, Hao Song, Minglu Li, Chen Cai, and Jianping Li. Modeling risk factors dependence using copula method for assessing software schedule risk. In *Software Engineering and Data Mining (SEDM), 2010 2nd International Conference on*, pages 571–574, June 2010.

# Attachment D4.2.5

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

Published in The 11th International Conference on Artificial Intelligence Applications and Innovations (AIAI'15), IFIP Advances in Information and Communication Technology, Vol. 458, pages 478–493, Springer, 2015

# 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.

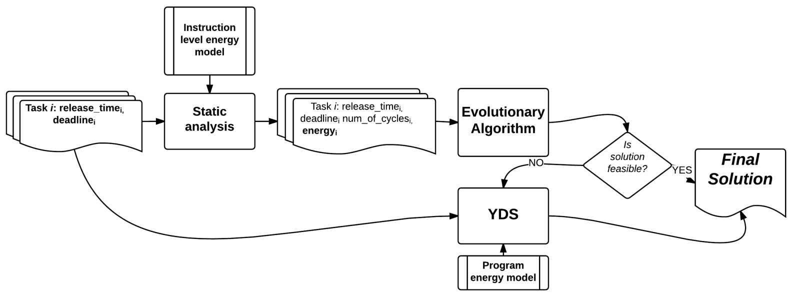

Instruction level energy model
Task i: release_time$_i$, deadline$_i$
Static analysis
Task i: release_time$_i$, deadline$_i$ num_of_cycles$_i$, energy$_i$
Evolutionary Algorithm
Is solution feasible?
YES
NO
Final Solution
YDS
Program energy model

**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.

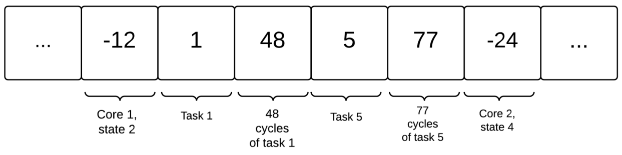


**Fig. 2.** An example of (part of) a solution (i.e., individual) representation.

**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 $P1$.

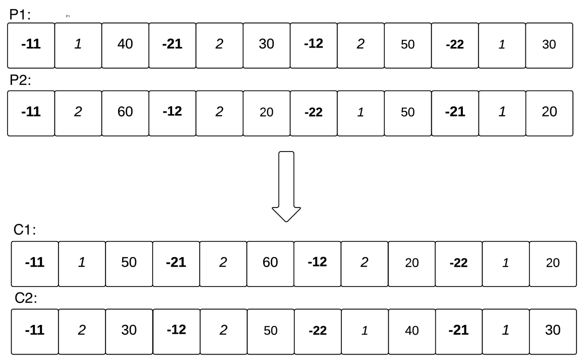
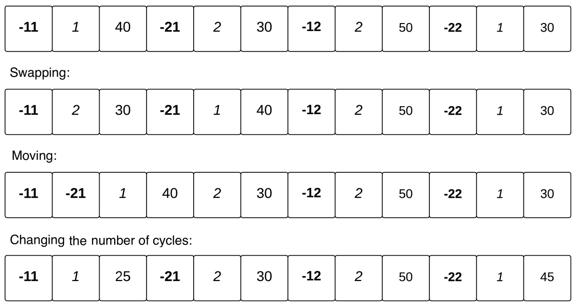


**Fig. 3.** An example of a crossover operation.

**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.



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **-11** | *1* | 40 | **-21** | *2* | 30 | **-12** | *2* | 50 | **-22** | *1* | 30 |

Swapping:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **-11** | *2* | 30 | **-21** | *1* | 40 | **-12** | *2* | 50 | **-22** | *1* | 30 |

Moving:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **-11** | **-21** | *1* | 40 | *2* | 30 | **-12** | *2* | 50 | **-22** | *1* | 30 |

Changing the number of cycles:

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

**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 $\omega_i$ is divided in two parts, $\omega_{i1}$ and $\omega_{i2}$, 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 $\omega_{i1}$ and $\omega_{i2}$ are calculated by solving the following system of equations:

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

**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).

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

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

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* project, and the Madrid M141047003 *N-GREENS* program.

## 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.

# Attachment D4.2.6

## Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable XMOS Chips

**Published in Hybrid Artificial Intelligent Systems (HAIS 2013), Lecture Notes in Computer Science, Vol. 8073, pages 401–410, Springer, 2013**

# Genetic Algorithm-based Allocation and Scheduling for Voltage and Frequency Scalable XMOS Chips

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

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

**Abstract.** In this work we present a novel approach, based on genetic algorithms, for automatic scheduling and allocation of tasks in a multi-processor multi-threaded architecture, together with an assignment of the appropriate voltage and frequency of each processor in a way the overall energy consumption is optimized and all task deadlines are met. The approach deals with scheduling, allocation and voltage and frequency assignment at the same time, and provides good solutions in a very short time. As far as we know, this is the first approach that supports two levels of parallelism: multi-processor and multi-thread.

## 1 Introduction

Dynamic power consumption due to switching activity in digital CMOS circuits can be expressed with the following formula: $P = \alpha C_{eff} V^2 f$, where $C_{eff}$ is the effective capacitance, $V$ is the operating voltage, $f$ is the operating frequency, and $\alpha$ the switching factor. If we can decrease the voltage supply and the operating frequency, the dynamic power will decrease significantly. On the other hand, static power which is the result of the leakage currents also decreases quadratically with voltage [?]. Thus, voltage decrease can achieve significant power and energy savings. This process is known as Dynamic Voltage and Frequency Scaling (DVFS). However, it slows down the operation of the circuit, and has to be applied in a way the required deadlines are still fulfilled. Furthermore, the process introduces additional latencies, so we have to develop a set of requirements that define the applicability of this approach.

The objective of this work is to optimize energy consumption through optimal scheduling and allocation for a set of tasks on XMOS chips, which are multiprocessor and multithreaded voltage and frequency scalable architecture. In XMOS chips threads are pipelined in a four-stage pipeline, where in each stage one instruction from different thread is executed, so in essence we can say that the threads also run in parallel. Thus, we deal here with two levels of parallelism. We assume that different processors can have different $(V, f)$ setting, while the threads running on the same processor at the same time must have the same $(V, f)$ setting.

Given a set of tasks and their corresponding deadlines, the objective is to provide a scheduling and allocation, and also assign voltage and frequency for each processor that would optimize the energy, while respecting the deadlines. The tasks are heterogeneous, and they in general have different starting time and deadline. We assume that there is no precedence between tasks, and no preemption. In order to solve the problem, we need to have safe estimates of power consumption of each task, as well as its execution time. Since this work falls into ENTRA project [**?**], whose main task is to provide the programmer the estimation of the energy consumption of his/hers program at compile time, we assume that there exists an analysis that would give us this information, as necessary input. On the other hand, there is a great body of work about time analysis, so we assume that the analyzer will provide us this information as well.

The general problem of scheduling and allocation is NP-hard. In order to solve it, different heuristic algorithms have been developed since they are capable of obtaining sub-optimal solutions in real time. Many of them use genetic algorithm (GA) [**?**,**?**,**?**] due to its fast exploration of the search space, which allows it to quickly find acceptable solutions. For this reason, our scheduler will also be based on GA. We will provide appropriate solution representation that captures the two levels of parallelism, i.e. at both processor and thread level, and in the same run performs allocation and scheduling and identifies appropriate $(V, f)$ setting in real time. As far as we know, this is the only solution for this type of problems.

The rest of the work is organized as follows. Section 2 details the sources of power consumption, while Section 3 explains the problem that is being solved and draws the constraints that are the basis for generating the solution. Section 4 details the implemented solution, while Section 5 explains the experimentation environment and presents the most significant results. Section 6 presents the most relevant related work, and finally, Section 7 draws the most important conclusions.

## 2 CPU Power Consumption

The energy required to complete a (set of) task(s) in time $T$ on one processor, given the frequency $f$ and voltage $V$ is defined by:

$$E_{cpu,f,V} = \int_{t_0}^{t_0+T} P_{cpu,f,V}(t)\mathrm{d}t \tag{1}$$

where $P_{cpu,f,V}$ is the time varying XCore power at *(V,f)* setting. This power can be calculated as:

$$P_{cpu,f,V}(t) = P^{fix}_{cpu,V} + P_{idle,f,V} + P^{act}_{cpu,f,V}(t) \tag{2}$$

where $P^{fix}_{cpu}$ is the portion of the power that includes PLL and leakage [**?**], which is the part that only depends on voltage, not on frequency. $P_{idle,f,V}$ is the power spent when the processor is not executing any application. For a certain fixed *(V,f)* setting, the sum of these two does not change in time, so in the further text we will call it standing power consumption, $P^{std}_{cpu,V,f}$. This power can be easily obtained by measuring the CPU power when there are no running applications for each *(V,f)* setting. On the other hand, $P^{act}_{cpu,f,V}(t)$ is the active power spent on switching activity during the execution of the application(s). Finally, we can write:

$$P_{cpu,f,V}(t) = P^{std}_{cpu,f,V} + P^{act}_{cpu,f,V}(t) \tag{3}$$

which put in 1 gives the energy consumed during time $T$ :

$$E_{cpu,f,V} = P^{std}_{cpu,f,V}T + \sum_{i=1}^{M} P_{i,f,V}T_i \tag{4}$$

where $P_{i,f,V}$ is the power spent by the application $i$, which is executed during time of $T_i$, and $M$ is the number of threads, i.e. the maximal number of applications that can be executed on one processor at certain moment. In the cases when the threads can finish more than one application within time $T$, formula 4 would have the following form:

$$E_{cpu,f,V} = P^{std}_{cpu,f,V}T + \sum_{i=1}^{M}\sum_{j=1}^{K} P_{ij,f,V}T_{ij} \tag{5}$$

where $K$ is the maximal number of applications a thread can execute in time $T$.

## 3 Problem Description

**Problem Definition**
Given a set of concrete tasks, provide optimal scheduling and *(V,f)* pair(s) for each processor in order to optimize energy consumption.

**Input**
- Set of tasks with their corresponding deadlines.
- Set of possible *(V,f)* pairs.
- Available hardware: $n$ - number of processors, $m$ - number of threads per processor.

**Output**
Viable scheduling and allocation that optimizes energy.

In the following text we assume the notation where variables are expressed using upper case letters, while constants are expressed using lower case letters.

### 3.1 Timing Constraint

In general case, for each new frequency $F_{new,i}$ of each processor $i$, the following should remain valid:

$$\forall i \in [1,n], \forall j \in [1,m],\ T_{oh,i} + \frac{C_{ij}}{F_{new,i}} \leq D_{ij} \tag{6}$$

where $T_{oh}$ is the time overhead introduced by DVFS and $C_{ij}$ is the number of clock cycles needed to execute the application $j$ on processor $i$, giving its execution time to be $\frac{C_{ij}}{F_{new,i}}$. This is reasonable to assume, given that in XMOS there are no pipeline stalls, nor cache misses, since there is no cache memory. We further have:

$$\forall i \in [1,n],\ T_{oh,i} = t_{oh_V} + T_{oh_{f,i}} \approx t_{oh_V} + \frac{10}{F_{old,i}} + \frac{2}{F_{new,i}} \tag{7}$$

where $t_{oh_V}$ is the time overhead of performing voltage scaling (assumed to be constant), while $T_{oh_f}$ if the time overhead of performing frequency scaling, which takes 10 clock cycles at most of the old clock, and two cycles of the new clock [**?**]. Finally, from 6 and 7 we get the timing constraints set:

$$\forall i \in [1,n], \forall j \in [1,m],\ F_{new,i} \cdot (C_{ij} + 2) \leq D_{ij} - t_{oh_V} - 10/F_{old,i} \tag{8}$$

where we consider that we know $t_{oh_V}$, and both $F_{old}$ and $F_{new}$ can take one value from the finite set of the pre-established values *(V,f)*.

## 3.2 Energy Minimization Constraint

The second set of requirements is derived from the condition of reducing the total energy during some known time $t$, high enough so that it permits the termination of all the applications. This implies the following condition:

$$\forall i \in [1,n], \forall j \in [1,m],\ t \geq \max_{i,j} D_{ij} \tag{9}$$

Thus, for each processor, we have:

$$\begin{aligned}
&\sum_{i=1}^{n} E_{old} \geq \sum_{i=1}^{n} E_{new} \Rightarrow \\
&\sum_{i=1}^{n} p^{std}_{i,cpu,F_{old,i},V_{old,i}} \cdot t + \sum_{i=1}^{n}\sum_{j=1}^{m} p_{ij,V_{old,i},F_{old,i}} \cdot \frac{C_{ij}}{F_{old,i}} \geq \\
&\sum_{i=1}^{n} e_{oh} + \sum_{i=1}^{n} p^{std}_{i,cpu,F_{new,i},V_{new,i}} \cdot t + \sum_{i=1}^{n}\sum_{j=1}^{m} p_{ij,V_{new,i},F_{new,i}} \cdot \frac{C_{ij}}{F_{new,i}}
\end{aligned} \tag{10}$$

where $e_{oh}$ is the energy spent on voltage and frequency scaling, $p_{ij,V_{old,i},F_{old,i}}$ and $p_{ij,V_{new,i},F_{new,i}}$ are estimated total power consumptions of the application $j$ on XCore $i$ in the different $(V,f)$ settings, while $p^{std}_{cpu}$ is the standing power explained in Section 2 in different settings. Finally, from 10, we get:

$$\begin{aligned}
&\sum_{i=1}^{n}\sum_{j=1}^{m}\left(\frac{p_{ij,V_{new,i},F_{new,i}}}{F_{new,i}} - \frac{p_{ij,V_{old,i},F_{old,i}}}{F_{old,i}}\right) \cdot C_{ij} \leq \\
&t \cdot \sum_{i=1}^{n}(p^{std}_{i,cpu,F_{old,i},V_{old,i}} - p^{std}_{i,cpu,F_{new,i},V_{new,i}}) - n \cdot e_{oh}
\end{aligned} \tag{11}$$

where the only unknown parameters are $C_{ij}$.

## 4 Proposed Solution

Our solution for optimal scheduling and allocation is based on GA. We have used steady-state GA, where the number of individuals of the population is the same in every generation and in every generation 60% of the population with lowest objective values is replaced with newly created individuals. Custom roulette wheel selector is used for the selection process. In the following text we will explain other important aspects of its implementation in more detail.

**Individual.** The starting point, and one of the most important parts, in designing a GA-based solution is always a representation of a solution, i.e. individual. In our case, the solution contains information about temporal and spatial allocation of each task. In other words, for each processor and each of its threads we should have an ordered (in time) set of tasks. However, since in this work we deal with DVFS, we have to add the information about the $(V, f)$ state of each processor. All the threads on the same processor have the same $(V, f)$ setting in the same moment, but we assume that different processors can have different $(V, f)$ setting, in order to solve the most general problem.

We can look at a solution to the scheduling problem as a permutation of the task identifiers, where their order also stands for the order of their temporal execution, assuming that each task has a unique identifier. On the other hand, in order to solve the allocation problem, i.e. on which thread (and which processor) each task is executed, we can add delimiters to the permutation of the task IDs that would define where the tasks are being executed, i.e. processor, thread and $(V, f)$ setting (the tasks between two delimiters are executed on the right-side one). In order to be able to distinguish delimiters from the task, they are used as negative three-digit numbers, where the first digit stands for the processor, the second for the thread on that processor, and the third for the processor $(V, f)$ setting (there is a finite number of settings). Part of a solution is depicted in Fig. 1, where tasks with IDs 1, 2, 5 and 7 are executed in that order on the thread 4 of the core 2, with the $(V, f)$ setting marked as 4. In the most general case, the order of delimiters is random. However, if two consecutive delimiters that belong to the same processor have different settings, this means that they are not being executed in parallel, since the state has to be changed. Representing solution in the described way has provided us with a relatively simple solution, which will not introduce great overhead when executing GA.

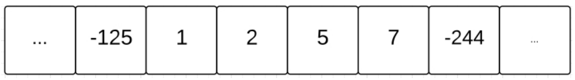


**Fig. 1.** Solution Representation

**Population Initialization.** We have used a heuristics when initializing the population in order to provide some good quality individuals from the beginning. According to it, the task is added to the thread in the way the total resulting energy up to the moment is minimal. However, the total energy is calculated for the time equal to the farthest deadline for each thread. In this way, more weight is given to the static power overhead. Thus, the objective of this heuristic is to promote delaying the execution of each task towards its deadline through minimizing the energy overhead. However, since in general GAs benefits from great variety of solutions, we also introduce random solutions. During the initialization process, each individual randomly chooses between heuristics and a randomly generated solution, where the heuristics has slightly bigger possibility to be chosen (0.6).

**Solution Perturbations.** Given that all the tasks and all the delimiters are different, different solutions are always a permutation of a set of tasks and the set of delimiters. This gives us the opportunity to use some of the permutation-based crossover operator, and in this case we are using the partial match crossover, since it performed better in the terms of objective function than the cycle crossover, and slightly better than order crossover in the terms of objective function and execution time. Since the order of delimiters is not important in the most general case, this operator provides at the same time variety in consecutive changes of $(V, f)$ settings, as well as moving tasks from one thread to another. Regarding mutation, it is implemented in the way that two random threads exchange two random tasks with a small probability.

**Objective Function.** Since the aim is to the minimize total energy, the objective function is the total energy consumed for executing the given set of tasks and it is calculated as presented in Section 2. However, the execution time for each thread is taken as the farthest deadline of its tasks, in order to take full advantage of the DVFS possibility. Furthermore, we have to be sure that the solution is viable, i.e. that all given deadlines are met. We deal with this problem through the penalization of the inviable solutions by multiplying their energy by 10. In this way, viable solutions will always have higher objective function and thus higher probability of surviving to the next generation.

## 5 Experimental Evaluation

### 5.1 Testing Environment

**XMOS Chips.** The target architecture for this work are XMOS chips. However, the same approach can be followed for any kind of DVFS-enabled multi-processor architectures. In the case of XMOS chip, both voltage and frequency scaling are possible and both introduce time overhead. All their chips provide the possibility of frequency scaling due to the existence of a programable frequency divider. The time overhead introduced when changing frequency is not more than 10 cycles of the old clock, plus two more cycles of the new clock.

On the other hand, only XS1-SU01A-FB96 [?] chip provides the possibility of voltage scaling due to the existence of two DC-DC converters whose output voltage can belongs to the range (0.6V, 1.3V). In order to apply DVSF, we should have a list of Voltage-Frequency *(V,f)* pairs or ranges that provide correct chip functioning. The latency in this case is not controllable, and can be estimated in the following way. Since the switching frequency of the converter is 1MHz, and assuming we have linear control, the bandwidth should be 1/10 to 1/7 of it, i.e. 150kHz in the best case. Thus, the time it takes for the output voltage to stabilize is 1/150kHz, which is around 6$\mu$s.

We have experimentally concluded that the XMOS chips can function properly in six $(V, f)$ settings given in the first two columns of Table 1. In order to include the possibility of shutdown, we could include the state (0(V),0(MHz)) and take the wake-up time as the latency of changing the state, and proceed in the same way. For the purpose of this experiment, we assume that we have four different processors, where each processor has eight different threads.

**Task Set** For the purpose of this initial experiment we have used the tasks from the well known SPECCPU2006 [?] benchmark. The input dataset is composed of 200 tasks randomly chosen, where each is one from the benchmarks. Each task is independent. The same reasonable deadline is assigned to each task, so it provides the possibility of applying DVFS. Their execution time is measured on a general purpose computer, and the execution time on an XMOS chip is estimated to be $T_{measured} \cdot \frac{f_{XMOS}}{f_{gp}}$. This estimation is based on the assumption that the total number of execution cycles is the same in both cases, and that it is representative of the total execution time. While this is true for the XMOS, in the general purpose computer this is not the case due to cache misses, pipeline stalls, etc. Thus, in the future we would have to profile the tasks on the XMOS chips, or use static analysis. It is important to point out that the duration of the tasks, as well as their energy, are much bigger than both time and energy overhead of DVFS scaling, so in this experiment the overhead will not be a limiting factor.

**Table 1.** Typical power consumption for each processor state

| $V(V)$ | $f(MHz)$ | $P_{dyn}(mW)$ | $P_{st}(mW)$ |
|---|---|---|---|
| 0.95 | 500 | 117.325 | 18.05 |
| 0.87 | 400 | 78.7176 | 15.138 |
| 0.8 | 300 | 49.92 | 12.8 |
| 0.8 | 150 | 24.96 | 12.8 |
| 0.75 | 100 | 14.625 | 11.25 |
| 0.7 | 50 | 6.37 | 9.8 |

Since this work represents an initial study of the approach, we have taken that the power consumption of each task is typical XMOS power consumption given in [?]. The estimations for different $(V, f)$ settings are estimated by scaling with voltage and frequency in the case of dynamic power, while the static power is scaled with voltage, i.e. $P_{dyn} = \frac{P_{dyn}^{base} \cdot f_{new} \cdot V_{new}^2}{f_{base} \cdot V_{base}^2}$ and $P_{st} = \frac{P_{st}^{base} \cdot V_{new}^2}{V_{base}^2}$. These values are given in Table 1 for each $(V, f)$ setting. However, it is assumed that in the future the analyzer will give us an estimation of power consumption of each task.

### 5.2 Obtained Results and Discussion

Genetic algorithm is executed on 500 individuals, during 100 generations. Greater number of individuals does not provide significantly better solution. In Fig. 2 we can see that the best objective value does not significantly change during the last iterations. The objective value is given in Wh. Under these settings, the total execution time of the algorithm is around six minutes on an Intel Dual Core machine, with 2.4GHz clock.

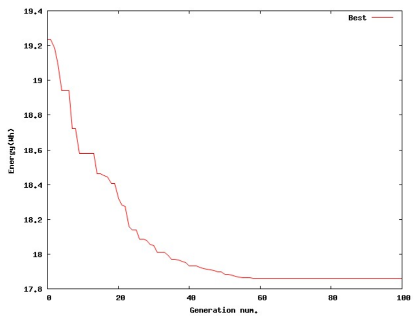


**Fig. 2.** Evolution of the best, the average and the maximum objective value

The average savings achieved in this way are 33.94%, with standard deviation of 0.56%, compared to the same scheduling and allocation without the DVFS. Speaking in the terms of statistical significance, we can be 95% sure that the obtained savings will belong to the interval (33.02%, 34.86%). A typical solution is presented in Fig. 3. Separate $(V, f)$ settings are distinguished with different colors, where the settings 1-6 correspond to the ones given in Table 1, and one time unit corresponds to one task. As we can observe, the majority of the tasks are executed in low power settings 4, 5 and 6.

## 6 Related Work

Since DVFS can provide significant energy savings, its optimal usage has been extensively studied. Some examples divide scheduling and allocation in two sep-

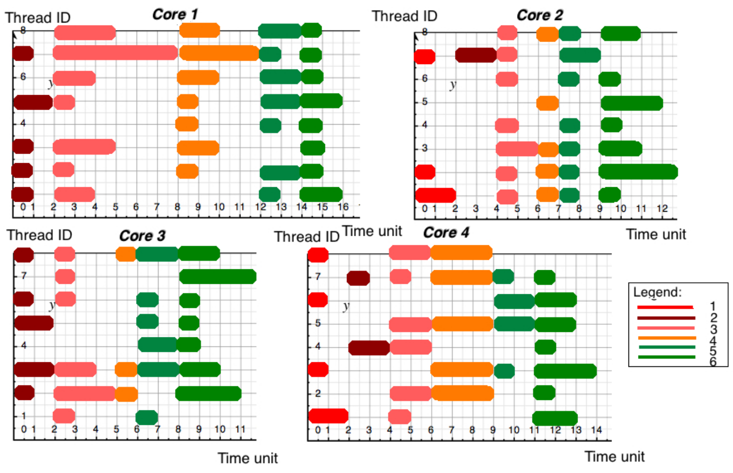


**Fig. 3.** A Scheduling and Allocation Solution per Core with Assigned DVFS setting

arate tasks, such as the one given in [?], where in the first step the allocation problem is solved using Linear Programming, while in the second the scheduling problem is solved for separate processors using Bin Packing. Another solution [?] solves the scheduling problem using GA, while it integrates DVFS in the fitness function. However, we believe that more optimal solutions could be achieved when solving scheduling and allocation at the same time, while also accounting for the DVFS. There is one example of GA-based scheduling [?] that combines scheduling, allocation and power management in one task. However, it deals only with voltage scaling.

There is also a significant group of publications on using GAs for optimal scheduling and allocation in multiprocessor systems with DVFS possibility. An example given in [?] treats the problem as bi-objective, where they want to minimize both energy and make span. The same objective is solved in another work [?], but using the island model of parallel GA populations. Yet, in this work our aim is to optimize the energy while meeting the deadlines, but our approach can easily be adapted to work as multi-objective. Another solution [?] treats the problem from two opposite points of view: in the first one, optimizes the energy given the scheduler length, while in the other optimizes the scheduling length given the energy bound. Finally, none of the solutions does not include the possibility of two levels of parallelism, where each processor can have a number of different threads executing in parallel.

## 7 Conclusions

In this work we have presented a solution for optimizing energy consumption for a multiprocessor and multithreaded architecture. The solution performs scheduling and allocation as one task, and deals with two levels of parallelism, which is the only solution of this kind as far as we know.

The solution will form part of the tool developed within the ENTRA project, when we will be able to include the power and time estimates provided by the ENTRA static analyzer. There are also possibilities to further improve the performances of the solution. For example, XMOS chips have the possibility to automatically reduce the frequency of the processor if all of its threads are waiting for an event, and in this way decrease the energy consumption even further. This feature will be included into future versions of our scheduler, as well as the possibility of shutting off separate components while they are not active.

**Acknowledgements.** The research leading to these results has been supported by the European FP7/2007-2013 318337 *ENTRA* project, and the Spanish TIN2012-39391-C04-01 *STRONGSOFT* project. svn st

## References

# Attachment D4.2.7

## An Energy-Aware Programming Approach for Mobile Application Development Guided by a Fine-Grained Energy Model

**Roskilde University, February 2016. submitted for publication**

# An Energy-Aware Programming Approach for Mobile Application Development Guided by a Fine-Grained Energy Model

Xueliang Li    John P. Gallagher

*Abstract*—**Energy efficiency has a significant influence on user experience of battery-driven devices such as smartphones and tablets. It is shown that software optimization plays an important role in reducing energy consumption of system. However, in mobile devices, the conventional nature of compiler considers not only energy-efficiency but also limited memory usage and real-time response to user inputs, which largely limits the compiler's positive impact on energy-saving. As a result, the code optimization relies more on developers. In this paper, we propose an energy-aware programming approach, which is guided by an operation-based source-code-level energy model. And this approach is placed at the end of software engineering life cycle to avoid distracting developers from guaranteeing the correctness of system. The experimental result shows that our approach is able to save from 6.4% to 50.2% of the overall energy consumption depending on different scenarios.**

## I. Introduction

The smartphone, the most popular mobile device, has been considered as one of the most important invention in the contemporary age. In February 2015, the penetration of smartphones was about 75% in the U.S [4]. This figure is still growing. With the improvement of hardware processing capability and software development environment, the smartphone is no longer a handset only to make phone calls, also play entertaining games, watch movie videos, browse web pages and so on. On the other hand, users are meanwhile frustrated by the limited battery capacity – applications running in parallel could easily drain a fully-charged battery within 24 hours.

Software optimization barely by the compiler achieves very little energy-saving for mobile devices since besides energy-saving the compiler on a mobile device has to think of many other important factors, such as limited memory usage and quick responses to user interactions. The Android platform, say, employs the Just-In-Time (JIT) compiler [6], also known as the dynamic compiler. Its optimization window is generally as small as one or two basic blocks in order to use less memory and quicken the delivery of performance boost. However, the small window largely restricts the space of energy-saving strategies. Eventually, the refactoring of code should rely more on developers.

Unfortunately, current software development is performed in an energy-oblivious manner. Throughout the engineering life cycle, most developers and designers are blind to the energy usage of code written by themselves. However, developers are desperate for the knowledge on energy-aware programming techniques. In the most popular software development forum StackOverFlow [40], energy-related questions are marked as favorites 3.89 more often than the average questions [34]. And among the energy-related questions, code-design-related ones are prominently more popular. Moreover, it has been estimated that energy-saving by a factor of as much as three to five could be achieved solely by software optimization [12]. To realize this, the first step is to analyze the energy accounting of source code at different levels of granularity and from different points of view.

In order to enable energy accounting of code, energy modeling of code is needed to bridge the gap between high-level source code and low-level hardware, where energy is consumed. However, traditional bottom-to-top modeling techniques [8], [36], [42], [44] face obstacles when the software stack of the system consists of a number of abstract layers. On the Android platform, say, the source code is in Java and then translated to Java byte-code, further to Dalvik [3] byte-code, native code and machine code and finally has chance to execute on the processors and consume energy. Consequently, the modeling task has to characterize the links among all the layers.

Instead of building a software energy model layer by layer, another approach to acquiring software-level energy information is to use the hardware readings, like CPU state residency, CPU utilization, L1/L2 Cache misses and battery trace, as predictors of software energy use [11], [33], [45], [46]. However, they are only capable of obtaining energy information at a coarse level of granularity such as methods or applications. Two pieces of work [14], [20] result in source-line energy information. The former requires low-level energy profiles. The latter employs accurate measurement to acquire the energy consumption of source lines.

The energy information on blocks or more coarse-grained units could identify the hot spots in the code, but it gives few clues about how to make changes to the code. The source line is also not an appropriate level of granularity to provide energy information. For instance, the header of `for` loop contains three segments which are *initialization*, *boolean* and *update* at the same source line, but usually have distinct numbers of executions. So the energy information about the source line of the header is not very sensible for developers.

Li et al. [21] propose a source-level energy model based on "energy operations", which is more fine-grained and able to provide more valuable information for code optimization. Compared with coarse-grained techniques, there are several advantages of the operation-based model in guiding the energy-aware programming techniques:

- The energy operations are basic units that constitute the energy consumption of entire software. Thus using the energy estimate of operations, the developers can assess the effects of code changes on energy consumption of code.
- It provides more valuable information on how to make changes. For example, the experiment shows that the "methode invocation" is the most expensive operation, suggesting that in some case we may inline some thin methods at the cost of losing the integrity of the structure of code.

In this paper, we propose an energy-aware programming approach guided by fine-grained energy model of source code. The generic procedures of the approach are as following:

- We utilize the methodology described in [21] to construct the operation-based source-code-level energy model, which is achieved by analyzing the data produced in a range of well-designed execution cases .
- The model generates energy accounting at operation and block level, which captures the energy characteristics of the code.
- We put efforts on the most costly blocks, where we refactor the code to remove, reduce or replace the expensive operations, meanwhile maintain its logical consistency with the original code.

Our target platform is an Android development board with two ARM quad-core CPUs, and the source code in our study is a game engine used in games, demos and other interactive applications. We evaluate the approach in three game scenarios, and the experimental result shows that it can save energy consumption by from 6.4% to 50.2% depending on different scenarios.

In the rest of this paper, we firstly introduce the identification of energy operations in Section II. The architectural setup and the design of execution cases are detailed in Section III. We elaborate the data collection and the model construction separately in Section IV and Section V, based on which we are able to capture the energy characteristics and optimize the source code in three different scenarios, `Click & Move`, `Orbit` and `Waves`, as respectively seen in Section VI, VII and VIII.

TABLE I: Examples of Energy Operations

| Operation | *Identified where:* |
|---|---|
| Method Invocation | *one method is called* |
| Parameter_Object | *Object is one parameter of the method* |
| Return_Object | *the method returns an Object* |
| Addition_int_int | *addition's operands are integers* |
| Multi_float_float | *multiplication's operands are floats* |
| Increment | *symbol "++" appears in code* |
| And | *symbol "&&" appears in code* |
| Less_int_float | *"<"'s operands are integer and float* |
| Equal_Object_null | *"=="'s operands are Object and null* |
| Declaration_int | *one integer is declared* |
| Assign_Object_null | *assignment's operands are Object and null* |
| Assign_char[]_char[] | *assignment's operands are arrays of chars* |
| Array Reference | *one array element is referred* |
| Block Goto | *the code execution goes to a new block* |

## II. Basic Energy Operations

There are two reasons why Li et al. [21] choose to build the source code energy model based on "energy operations". Firstly, an energy operation is "atomic", which means that all the statements, source lines, blocks and methods are made up of a certain number of kinds of operations (in the experiment, we have 120 operations). Secondly, it is fine-grained. Energy information at the level of source lines or methods is useful; however, information at source line level could not distinguish energy consumption of two operations in the same source line, for example.

Energy operations are identified directly from source code. The enumeration of the operations is inspired by Java semantics [7], which specifies the operational meaning, or behavior, of the Java language, which is the target language in the experiment. We intuitively identify semantic operations that perform operations on the state and may be energy-consuming, and let them be our energy operations. Ones that have little or no energy effect will automatically be identified by the regression analysis in the later stage of the analysis. Table I lists 14 representative operations out of a total of 120 in the experiment. They include arithmetic calculations like *Multi_float_float*, *Addition_int_int*, in which operands types are explicit, as well as *Increment* whose operand is implicitly an integer. Boolean operations and comparisons, such as *And*, *Less_int_float* and *Equal_Object_null* also form one major part. *Method Invocation* and *Block Goto* are important for the control flow which plays a key role in the execution of the code. Assignments and *Array Reference* will unexpectedly take a significant amount of the application's energy consumption, as will be shown in Section VI-A.

TABLE II: Examples of Library Functions

| Class | Function |
|---|---|
| ArrayList | *add, get, size, isEmpty, remove* |
| GL10 | *glBindTexture, glDisableClientState*<br>*glDrawElements, glEnableClientState*<br>*glMultMatrixf, glTexCoordPointer*<br>*glPopMatrix, glPushMatrix*<br>*glTexParameterx, glVertexPointer* |
| Math | max, pow, sqrt, random |
| FloatBuffer | *position, put* |

The application also employs a diversity of library functions that may be written in different languages and at lower levels of the software stack. On the other hand, usually a limited number (67 in the experiment) of library functions are frequently called in one application. So we treat them as basic modeling units. The examples of highly-used library functions in the experiment are shown in Table II. For instance, the functions in the class of *GL10* are responsible for graphic computing.

## III. Experimental Setup

In this section, we will introduce the setup of the target device and source code. We also explain the design principles of the execution cases.

### A. Target Device

Experimental target: we employ an Odroid-XU+E development board [30] as the target device. It possesses two ARM quad-core CPUs, which are Cortex-A15 with 2.0 GHz clock rate and Cortex-A7 with 1.5 GHz. The eight cores are logically grouped into four pairs. Each pair consists of one big and one small core. So from the operating system's point of view there are four logic cores. In our experiment, we turn off the small cores and run workload on big cores at a fixed clock frequency of 1.1 GHz. We do this in order to remove the influence of voltage, clock rate and CPU performance on the power usage.

Power Reading Script: Odroid-XU+E has a built-in power monitoring tool to measure the voltage and current of CPUs with a frequency of 30 Hz and updates the samples in a log file. We wrote a script to obtain the samples from the file. During execution we run the script on an idle core to minimize its influence on the application.

Note that the power monitor gives two sequences of power samples: one is for the big cores and the other is for the small cores. We pick the sequence of power samples of the big cores, because we only run workload on them.

### B. Target Source Code

The target source code is the Cocos2d-Android [2] game engine, a framework for building games, demos and other interactive applications. It also implements a fully-featured physics engine. Games are increasingly popular on mobile phones, and the applications include more and more fancy and energy-consuming features, requiring high CPU performance. Energy modeling and accounting, explained in the rest of this paper, will present opportunities to guide software development towards energy efficiency.

### C. Design of Execution Cases

The execution cases whose energy usage is measured and analyzed represent typical sequences of actions during game, including user inputs. We focus on three scenarios which are `Click & Move`, `Orbit` and `Waves`.

In the `Click & Move` scenario, the sprite (the character in the game) moves to the position where the tap occurs. In the `Orbit` scenario, the sprite together with the grid background spins in the three-dimension space. In the `Waves` scenario, the sprite scales up and down, meanwhile the grid background waves like flow. In both the `Orbit` and `Waves` scenarios, the animation will restart from the starting point whenever and wherever the tap occurs.

To simulate the game scenarios under different sequences of user inputs, we script with the Android Debug Bridge [1] (ADB) , a command line tool connecting the target device to the host, to automatically feed the input sequences to the target device.

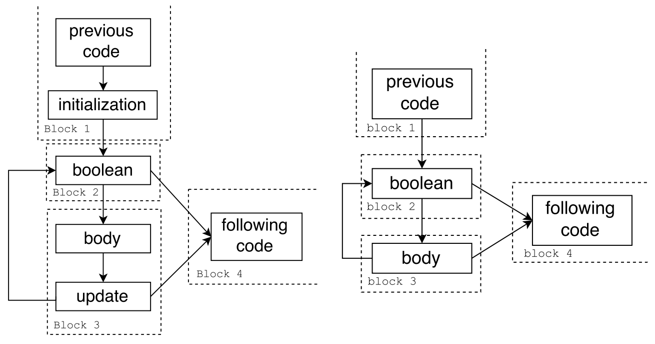
In order to obtain a more varied set of execution cases, we vary the executions of individual basic blocks in the code. This is achieved by systematically removing a set of blocks for each execution case, using the control flow graph obtained using the Soot tool [38]. We ensure that each block could be removed in some execution case. Thus an execution case is made up of one user input sequence and one set of basic blocks.

## IV. Data Collection

In this section, we describe the collection of data on the number of times each operation executes and the energy consumption of an execution case, based on which we construct the energy model.

### A. Number of Executions of Operations

To obtain the number of times that each operation executes in an execution case, we need to determine at which level of granularity to track the execution. We choose the level of "blocks". A block is a sequence of consecutive statements, without loops or branches. It is sufficient to track block executions, since if one part of a block is processed, the rest certainly will be processed as well.



(a) `for` loop (b) `while` loop

Fig. 1: Block division of `for` and `while` loops in control flow graph.

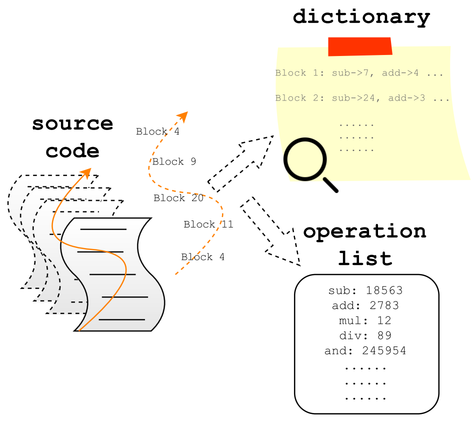


Fig. 2: The flow of the operation-execution data collection.

We could consider collecting data at other levels of granularity. Tracing individual statements might overload the capacity of the target device. On the other hand, methods or classes are unsuitable execution units, since we cannot determine which parts of the method or class will be active during the execution, and this information about energy operations is lost.

We then divide the source code into blocks. For individual syntactic structures, we deal with block division case by case. `For` loops and `while` loops are handled as shown in Figure 1. In a `for` loop, the header usually has three segments which are *initialization*, *boolean* and *update*. They are divided into three different blocks. Similarly, we set the `while` header itself as a block ("block 2" in Figure 1b). In order to build the log, we instrument the source code with a log instruction at the beginning of each block.

The generic view of the collection of the operation-execution data is displayed in Figure 2. We build a dictionary showing, for each block, the number of occurrences within it of each energy operation, such as those in Table I. This dictionary is built using a parser that traverses all the blocks in the code.

Then, using the log file recording the processed blocks, together with the dictionary, we can sum up the number of times that each energy operation is executed during an execution case. To be more precise, let $B_i$ be the number of times that the $i^{th}$ block is executed (this is obtained from the log file). Let $O_{i,j}$ be the number of occurrences of operation $j$ in block $i$ (this is obtained from the dictionary). Then the total number of executions of the $j^{th}$ operation is $\sum_{i=1}^{n}(B_i * O_{i,j})$, where $n$ is the total number of blocks.

### *B. Energy Approximation from Power Samples*

We write a script to obtain the power samples from the built-in measurement component with a frequency of 30 Hz. The power samples are the discrete values sampled from the power trace; we approximate energy consumption by calculating Equation (1): $p = power(t)$ is the power trace, that is, the continuous power-vs-time function; $power(t_i)$ is the power sample at time-stamp $t_i$; $\Delta_i$ equals to $t_i - t_{i-1}$, which is the interval between two sequential samples.

$$E = \int_{t_0}^{t_n} power(t)\,dt \approx \sum_{i=1}^{n} power(t_i) \cdot \Delta_i \qquad (1)$$

$$where \quad t_0 \leq t_1 \leq t_2 \cdots \leq t_{n-1} \leq t_n$$

### *C. Challenges in Practice*

**Measurement limitation:** the sampling rate of the built-in power monitor is 30 Hz. However, the instruction execution rate is about several million per second. That means, one power sample measures the energy cost of hundreds of thousand instructions. Even though the state of the art of the power measurement can reach a sampling rate of tens of KHz [17], one power sample still includes up to thousands of instructions.

To deal with this problem, we first lengthen the sessions of all the execution cases to above 100 seconds, and then run each case for ten times to calculate their average energy cost. Compared with the execution cases that only run once with sessions around one second, this approach can reduce the error of measuring energy consumption of the code by three orders of magnitude.

**Run-time context:** during the running of the application, the Dalvik virtual machine performs garbage collection, which is not part of the application and still could be included in the power samples.

The Dalvik virtual machine produce time-stamp logs when launching the garbage collection procedure. We consider the garbage collection as one library function, so it will be integrated in the model.

**Code instrumentation and power reading script:** although the instrumentation is at block level rather than statement level, its impact on energy consumption is still not negligible and its cost is as much as 50% of the application's energy consumption itself. Also, the energy cost of the power reading script is up to 5% of the application's consumption.

We followed three experimental principles to address this problem. Firstly, for each execution case, the log of the execution path and of the power samples are separated into two separate runs. In the first round, we record the execution path without reading power samples. In the second round, we only trace power and disable the instrumented log instructions. So for each execution case, the instrumentation for logging the execution path will not influence the power samples.

Secondly, in each of the two runs, the main process of the application is allocated to one CPU core, while the thread logging execution path or power samples is allocated to another CPU core, minimizing effects due to interaction of the threads.

Thirdly, we design one "idle execution case" paired with each execution case; this only runs the power reading script without the application. By this means we can get the energy consumption of the main application process by excluding the cost of the "idle execution case" from the execution case. Note that the durations of execution cases are different, so we need to have a distinct "idle execution case" for each execution case.

In summary, each execution case will be run 21 times: once for tracing the execution path; ten times for calculating the average energy consumption of the "idle execution case", and ten times for calculating average energy consumption of the execution case.

## V. Model Construction

The entire energy consumption is composed of three parts: the cost of energy operations, the cost of library functions and the idle cost. The aimed model is formalized in Equation (2). The cost of energy operations is the sum of $Cost_{op_i} \cdot N_e(op_i)$ (the cost of one operation multiplied by the number of its executions), where $op_i \in EnergyOps$. $EnergyOps$ is the set containing all the operations. The cost of library functions is the sum of $Cost_{func_i} \cdot N_e(func_i)$ (the cost of one library function multiplied by the number of its executions), where $func_i \in LibFuncs$. $LibFuncs$ is the set of library functions. The $Idle\,Cost$ is the energy consumption of the "idle execution case". The lengths of case sessions are varying, so the $Idle\,Cost$ is different for each execution case.

$$E = \sum^{op_i \in EnergyOps} Cost_{op_i} \cdot N_e(op_i) \qquad (2)$$
$$+ \sum^{func_i \in LibFuncs} Cost_{func_i} \cdot N_e(func_i) + Idle\,Cost$$

The model construction is based on regression analysis, finding out the correlation between energy operations and their costs from the data obtained in the execution cases. We set out the collected data in the matrices in Equation (3). The leftmost matrix ($N$) contains the execution numbers of $l$ operations (including energy operations and library functions) in $m$ execution cases, acquired as shown in Section IV. Each row indicates one execution case. Each column represents one operation. The vector ($\vec{cost}$) in the middle contains the costs of $l$ operations, which are the values we are aiming to estimate. The vector ($\vec{e}$) on the right of the equal mark contains the measured entire energy costs of the execution cases. So for each execution case, the entire energy cost is the sum of the costs of operations. It should be noticed that the energy costs $\vec{e}$ exclude the $Idle\,Cost$ which is measured when no application workload is being processed.

$$\begin{pmatrix} n_1^{(1)} & n_2^{(1)} & \dots & n_l^{(1)} \\ n_1^{(2)} & n_2^{(2)} & \dots & n_l^{(2)} \\ & \dots & \dots & \\ n_1^{(m-1)} & n_2^{(m-1)} & \dots & n_l^{(m-1)} \\ n_1^{(m)} & n_2^{(m)} & \dots & n_l^{(m)} \end{pmatrix} \times \begin{pmatrix} cost_1 \\ cost_2 \\ \dots \\ cost_l \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ \dots \\ e_{m-1} \\ e_m \end{pmatrix} \qquad (3)$$

Inevitably, the power samples are not absolutely accurate. Furthermore, the energy model in reality is unlikely to be completely linear. For these reasons Equation (3) may be unsolvable, that is, the vector $\vec{e}$ is out of the column space of $N$. We thus employ the gradient descent algorithm [29] to compute the approximate values of $\vec{cost}$.

The elements of $\vec{cost}$ are randomly initialized and then improved by the gradient descent algorithm iteratively. We first introduce the error function $J$ (computed by Equation (4)) which indicates the quality of the model. The smaller $J$ is, the better the model is. $\vec{n^{(i)}}$ is the $i^{th}$ row in $N$, $\vec{cost}$ is the middle vector above. $\vec{n^{(i)}} \times \vec{cost}$ is the estimated energy cost for the $i^{th}$execution case, $e^{(i)}$ is its observed energy cost. $J$ first computes the sum of the squared values of the estimate errors of all the execution cases, which is afterwards divided by $2m$ to get the average

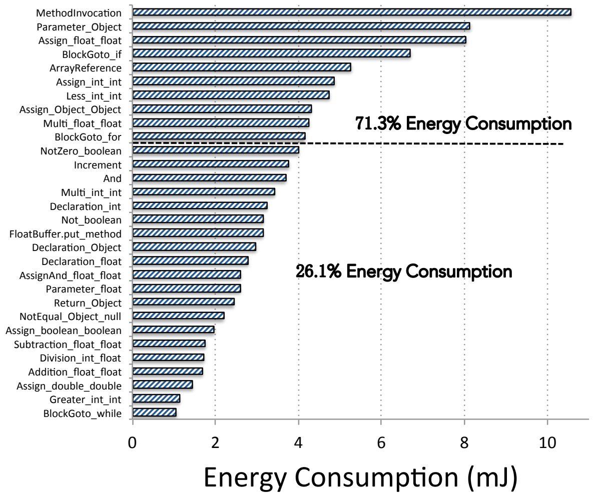


Fig. 3: The top 30 energy consuming operations in `Click & Move` scenario.

value.

$$J(cost_1, cost_2, ...cost_l) = \frac{1}{2m}\sum_{i=1}^{m}(\vec{n^{(i)}} \times \vec{cost} - e^{(i)})^2 \tag{4}$$

$$cost_j := cost_j - \alpha\frac{\partial J(cost_1, ...cost_j, ...cost_l)}{\partial cost_j} \tag{5}$$

$$= cost_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(\vec{n^{(i)}} \times \vec{cost}) \cdot n_j^{(i)}$$

$$j = 1, 2, ...l$$

The idea of gradient descent is to minimize $J$ by repeatedly updating all the elements in $\vec{cost}$ with Equation (5) until convergence. The partial derivative of the function $J$ on $cost_j$ gives the direction in which increasing or decreasing $cost_j$ will reduce $J$. Every element ($cost_j$) of $\vec{cost}$ is updated one by one in each iteration. The value $\alpha$ determines how large the step of each iteration is. If it is too large, the extremum value will possibly be missed; if too small, the minimizing process will be rather time-consuming. It needs to be manually tuned. Theoretically, the gradient descent algorithm could only find the local optima. In practice, we randomly set the values in $\vec{cost}$ and restart the entire gradient decent procedure for several times to look for the global optima.

In the experiment, the three scenarios (`Click & Move`, `Orbit` and `Waves`) separately have their own processes of data collection and model construction since different scenarios may have different sets of parameters (costs of operations) for the model (Equation (2)). The cost of the same operation is not absolutely constant in certain cases, one of the reasons is that the values of operands influence the energy consumption of operations, as seen in [31]. Our modeling approach is trying to make a good approximation of the costs of operations for individual scenarios.

TABLE III: NMAE in Cross Validation

| Scenario | Set | 1st | 2nd | 3rd | 4th |
|---|---|---|---|---|---|
| Click & Move | Training | 17.7% | 15.0% | 13.6% | 18.9% |
| | Validation | 14.2% | 14.2% | 19.7% | 17.8% |
| Orbit | Training | 19.9% | 17.9% | 14.4% | 16.8% |
| | Validation | 11.7% | 17.0% | 18.0% | 15.0% |
| Waves | Training | 13.9% | 14.1% | 14.8% | 15.0% |
| | Validation | 16.8% | 16.7% | 16.1% | 17.2% |

To validate the reliability of model, we apply the four-round cross validation. If the model is proved to be reliable, then we use it for the energy accounting in later stages, otherwise we try other solutions to improve the model. The four-round cross validation procedure is as following: the set of execution cases are randomly divided into four subsets; in each round, one of them is chosen to be the validation set and the others together to be the training set.

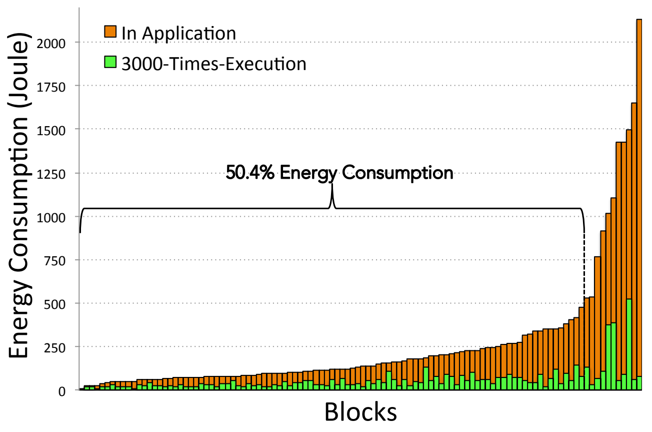
$$NMAE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\hat{e^{(i)}} - e^{(i)}}{e^{(i)}}\right| \tag{6}$$

In Table III, we can see the Normalized Mean Absolute Error (NMAE) of the model in three scenarios in training and validation sets in the four rounds. The NMAE is a well-known statistical criterion that shows how well the estimated value matches the measured one. It is computed by Equation (6), the mean value of normalized difference between the predicted energy cost $\hat{e}$ and the measured cost $e$. The lower the ratio the better the result. In the three scenarios, the NMAE in training sets ranges from 13.6% to 19.9%, and in validation sets from 11.7% to 19.7%.

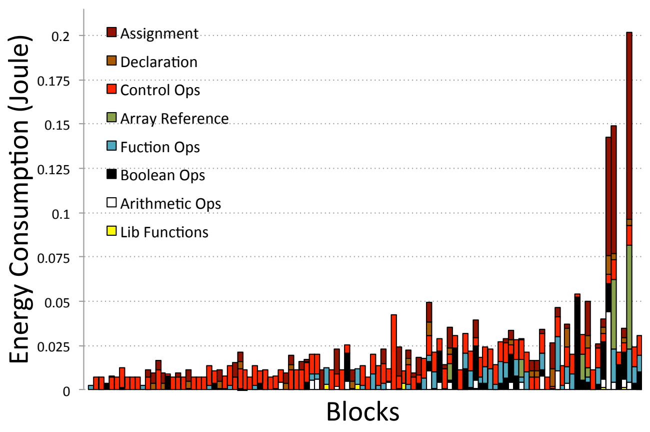
For the three scenarios, the sets of parameters respectively generated in the 2nd, 4th and 3rd rounds of cross validation are chosen to help analyze the energy property of the code in Section VI-A, because they have good balance on both training and validation sets. Their NMAEs are around 15.0%, which means the model's inference accuracy is around 85.0%.

## VI. The `Click & Move` Scenario

In this section, we detail energy accounting at operation and block level, according to which we improve the most costly blocks by removing, reducing or replacing the most expensive operations. Later in Section VII and Section VIII, when we talk



(a) Block costs "In Application" and at "3000-Times-Execution".



(b) Energy proportions of different kinds of operations in blocks.

Fig. 4: Energy distribution in `Click & Move`. Blocks are sorted by the order of their run-time energy costs "In Application".

about the `Orbit` and `Waves` scenarios, we will briefly introduce the energy characteristics of the code and use larger part for the code improvements.

## A. *Energy Accounting*

The energy model of app source code based on energy operations facilitates comprehensive energy accounting at different levels of granularity and from various viewpoints. In this section, we will see the rank of the most expensive operations, and the contributions of different operations to the energy consumption of each block.

**Operation Level:**

Figure 3 shows the top 30 energy consuming operations, which are ranked by their single-execution energy costs. "71.3% Energy Consumption" presents the percentage of sum of costs of top 10 operations in the total cost, considering their different numbers of executions in the `Click & Move` scenario. "26.1% Energy Consumption" means the percentage of operations from 11th to 30th. The percentages indicate that the energy-usage of the code is largely determined by a relatively small number of operations. It is because these operations are frequently used and meanwhile expensive themselves. The 30 operations out of 187 (including library functions) take up 97.4% of the whole cost of the code, in which the top 10 consumes the major part with a percentage of 71.3%.

Usually, it is supposed that the sophisticated arithmetic operations, such as multiplications and divisions, should be the most costly. However, the result shows that *Method Invocation* ranks the highest. This is due to a sequence of complex processes to fulfill *Method Invocation*, such as storing the return address and managing the stack frame. Instance methods are always implicitly passed a "this" reference as their first parameter. It suggests a trade-off between the structure and the energy saving when writing the code. That means, in certain cases, we could unpack some thin methods that are highly-invoked in the code, at the cost of losing the integrity of the structure of the code to some extent.

Unexpectedly, only one arithmetic operation, *Multi_float_float*, is a member of the top 10. And there are only six arithmetic operations in the top 30. They together cost only 6.1% of the overall energy consumption of the application, which is contrary to our instincts.

Later in block-level energy accounting, we will see that assignments, comparisons and *Array Reference* play significant roles in the overall energy consumption. This is not only because they are frequently used, but also because they are costly as operations themselves, as shown in Figure 3.

*Block Goto* operations are expensive as well. Based on the types of conditionals and loops where "Block Goto" occurs, they are classified into *BlockGoto_if*, *BlockGoto_for* and *BlockGoto_while*. The result shows that they cost different amounts of energy as operations themselves, respectively 6.7 mJ, 4.1 mJ, 1.1 mJ. And together with *Method Invocation*, they take up 37.6% of the total energy consumption of the application.

**Block Level:**

In the execution cases, we have 108 active blocks with a wide diversity of energy usage. As shown in Figure 4a, "In Application" here means running the `Click & Move` scenario with the full set of blocks. The costs of blocks "In Application" are plotted as orange bars. Note that, blocks here obviously have distinct execution times. The cost of a fixed number (3000) of executions of one block are calculated by multiplying its single-execution cost by 3000. This could help us compare the single-execution costs of different blocks. The costs of blocks at "3000-Times-Execution" are plotted as green bars.

Similar to energy distribution on operations, only a small number (11 blocks) of all the blocks uses up nearly half of the entire cost, which indicates that putting efforts on optimising a small group of blocks can achieve significant energy-saving.

There are two factors that make one block costly "In Application". The first factor is a large number of executions. For example, the most costly block "In Application" (the rightmost orange bar in Figure 4a) has a large number of execution times. This block takes only 30.6 mJ for single-execution but 2128.6 Joule when running "In Application". The second factor is the energy consumption of the block itself. For example, the three prominent green bars in Figure 4a, whose single-execution costs are 201.5 mJ, 146.9 mJ and 142.8 mJ. We will later zoom in these three blocks to see which operations contribute to their energy costs.

We can further observe the energy proportions of operations in each block in Figure 4b. To illustrate, operations are grouped into eight classes. Specifically, the "Block Goto" operations and *Method Invocation* are gathered in *Control Ops*; the parameter passing and the value returns of methods are in *Function Ops*; the comparisons and Booleans are in *Boolean Ops*; all the arithmetic computations are in *Arithmetic Ops*; all the library functions are in *Lib Functions*.

Most of the blocks cost less than 25 mJ for single-execution. In these blocks, *Control Ops* occupy the major part of the energy consumption, in contrast, *Arithmetic Ops* only take a tiny proportion.

For those three most prominent blocks, assignments and *Array Reference* are the biggest energy consumers. Furthermore one of the three blocks has the largest proportion of *Arithmetic Ops* among all the blocks.

The most expensive block "In Application" consists of three even parts: *Control Ops*, *Function Ops* and *Boolean Ops*. This block is the main entrance of the game engine to draw and display frames, so its works are conditional judgments and method invocations.

### B. Code Optimization

The most important consideration of app developers is to guarantee the correctness of software, which should then be followed by energy-efficiency. So our energy-aware programming approach is applied at the end of software engineering life circle when the software system is roughly complete.

The overview of energy-aware programming approach is firstly finding the most costly blocks, where we analyze the energy breakdown among the operations, and make changes to the code to remove, reduce or replace the costly operations.

TABLE IV: The top 10 costly blocks in `Click & Move`.

| Block ID | #Executions | Energy Cost (J) |
|---|---|---|
| CCNode.visit() | 19462 | 2128.6 |
| CCNode.transform() | 18903 | 1648.4 |
| CCTextureAtlas.putVertex() | 2119 | 1494.4 |
| CCNode.visit().if_4.for_1 | 16880 | 1426.8 |
| CCNode.transform().if_1 | 19664 | 1426.3 |
| CCTextureAtlas.putTexCoords() | 2120 | 1107.8 |
| CCAtlas.updateValues().for_1 | 2173 | 1018.7 |
| CCNode.visit().if_3.for_1 | 8356 | 915.7 |
| CCSprite.draw() | 8594 | 766.9 |
| CCTexture2D.name() | 13085 | 537.5 |

We look into the top 10 costly blocks "In Application" (see Table IV). For example, *CCNode.visit()* is the entrance block of the *visit()* function; *CCNode.visit().if_4.for_1* is the body block of the `for` loop. These 10 blocks are distributed in seven methods, so the code review does not require heavy labor. We find four easy optimization opportunities in blocks, such as *CCNode.visit()*, *CCNode.visit().if_4.for_1* and *CCTexture2D.name()*. There are also other opportunities in other blocks supposed to save energy, but requiring more efforts and gaining little. For example, *CCAtlas.updateValues().for_1* has several busy arithmetic expressions. Usually it is believed that replacing the busy expression with an variable could reduce energy cost, however in this case the overhead of variable declaration counteracts the energy-saving.

The four opportunities to improve the code are very simple and effective, but can only be discovered by the operation-level energy information. The changes will be shown as following.

**Program 1** Simplified parts of **original** code in *CCNode.visit()*

```
if (children_ != null) {
   if_body1;
}
draw(gl);
if (children_ != null) {
   if_body2;
}
```

**Program 2** The changed Program 1

```
if (children_ != null) {
   if_body1;
   draw(gl);
   if_body2;
} else {draw(gl);}
```

**If Combination:**

This change is made in the most costly block *CCNode.visit()*, which has two comparisons, two Boolean operations, one *Method Invocation* and one parameter passing. In fact, the two `if` headers make the same comparison, as shown in Program 1. We change the code to Program 2, which combines the two `if` statements and meanwhile keep it logically consistent with Program 1. By the means each execution of the block can reduce one comparison, and when the condition is false, it can additionally reduce one *BlockGoto_if* .

**Program 3** Simplified parts of **original** code in *CCNode* class

```
public void visit(GL10 gl) {
      ......
   transform(gl);
      ......
}
public void transform(GL10 gl) {
   tranform_body;
}
```

**Program 4** The changed Program 3

```
public void visit(GL10 gl) {
      ......
   transform_body;
      ......
}
public void transform(GL10 gl) {
   transform_body;
}
```

**Inner-Class Method Inline:**
When "In Application", the *transform()* function is invoked 18903 times and mostly by *visit()* function. We change the Program 3 to Program 4 by inserting the body of *transform()* into *visit()*, meanwhile remaining the original *transform()* function in case that other parts of the code call it. This change can largely decrease the number of *transform()*'s *Method Invocation*s that are very expensive. However, it may be at the cost of losing readability of the code, which could also be compensated by adding explanatory comments.

**Program 5** The full version of Program 2

```
if (children_ != null) {
for (int i=0; i<children_.size(); ++i) {
   CCNode child = children_.get(i);
   if (child.zOrder_ < 0) {
       child.visit(gl);
   } else
      break;
}
draw(gl);
for (int i=0; i<children_.size(); ++i) {
   CCNode child = children_.get(i);
   if (child.zOrder_ >= 0) {
      child.visit(gl);
   }
}
} else {draw(gl);}
```

**Loop-Invariant Code Motion:**
*CCNode.visit().if_3.for_1* and *CCNode.visit().if_4.for_1* are entrance blocks of the two `for` loops as seen in Program 5. These two loops have a quantity, *children_.size()*, which is computed in each iteration but the value is constant. We thus hoist it outside the loop, as shown in Program 6, which can vastly save the energy of invoking and executing the *size()* function during every iteration. At the same time, we move the declaration of the *child* outside the loop, considering the cost of *Declaration_Object* is about 2.97 mJ and also in the top 30.

**Program 6** The changed Program 5

```
CCNode child = new CCNode(); //added
int children_size = children_.size(); //added
if (children_ != null) {
   for (int i=0; i<children_size; ++i) { //changed
      child = children_.get(i); //changed
      if (child.zOrder_ < 0) {
         child.visit(gl);
      } else
          break;
   }
draw(gl);
for (int i=0; i<children_size; ++i) { //changed
   child = children_.get(i); //changed
   if (child.zOrder_ >= 0) {
      child.visit(gl);
   }
}
} else {draw(gl);}
```

**Inter-Class Method Inline:**
*CCTexture2D.name()* is the 10th costly block and costs 537.5 Joule "In Application". However, its job is to simply get the value of the private member variable, *_name*, of the class *CCTexture2D*. And this method has only two callers in the code. So we consider to make this variable public and let the two callers directly get access to the variable, which avoids the cost of *Method Invocation*. This change may harm the encapsulation of data, however, only one member of one class is changed. The trade-off between energy-saving and data encapsulation will be at last decided by developers.

## C. Evaluation

Figure 5 illustrates the energy consumption of the software without and with the changes introduced in the previous section. From left to right, the bars indicate accumulative effects of the changes. For example, "+ *If Comn*" is the energy consumption of the code with "If Combination"; "+ *Inner-Class MI*" is the energy consumption of the code with the changes of both "If Combination" and "Inner-Class Method Inline". Totally, these four simple changes save 6.4% of the entire energy consumption without influencing the functionality of code. These changes are made in the basic part of the game engine, where most applications will base on, so any gain here can have fundamental impact. Furthermore, these changes are made with little knowledge about the algorithm of code, the developers who wrote the code are surely able to improve the code much more and achieve more energy-saving.

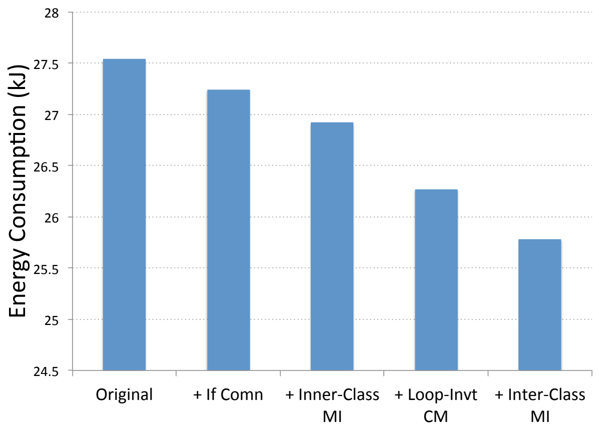


Fig. 5: Energy consumption of the code without and with the changes in `Click & Move`.

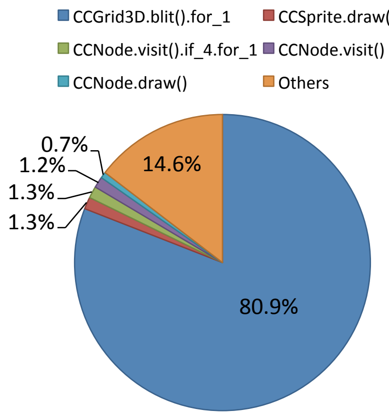


Fig. 6: In `Orbit` scenario, the energy proportions of blocks "In Application".

## VII. The Orbit Scenario

In this section, we briefly introduce the energy characteristics of `Orbit` scenario. Afterward, we improve the most costly blocks according to the types of expensive operations. In Section VII-C, we can see that the improvement can save as much as 50.2% of the overall energy consumption.

### A. Energy Accounting

In the `Orbit` scenario, the block *CCGrid3d.blit().for_1* dominates the overall energy consumption. As shown in Figure 6, 80.9% of the entire cost is consumed by this block. The second costly block consumes only 1.3%. "In Application" here means running the `Orbit` scenario without removing any block. Later in Section VII-B, we will barely put attention on this single block, requiring fairly little effort to achieve improvements.

### B. Code Optimization

Program 7 shows the original code of *CCGrid3D.blit().for_1*. In this block, the *Control Ops* (*BlockGoto_for* and *Field Reference*) use up 35.6% energy; *Boolean Ops* use up 20.5%; the assignments use up 16.7%; *Arithmetic Ops* use up 14.0%; *Lib Functions* use up 13.3%. We find three easy changes to reduce or replace the expensive operations.

**Loop-Invariant Code Motion:**

In this block, the value of *vertices.limit()* is constantly 2112, we thus hoist it outside the loop and replace it with the variable *limit*, as shown in Program 8. This change avoids calls of *vertices.limit()* and at the same time decreases a small amount of *Field Reference*.

**Loop Unrolling:**

Also as shown in Program 8, we duplicate the loop body eight times, which reduces the times of comparisons, *BlockGoto_for*s, assignments and additions. Note that, we set the value of increment as 24 since 24 is a factor of the *limit*, 2112.

**Full-Use of Library Function:**

The job of Program 7 or Program 8 is getting all the elements in *vertices* one by one and putting them into *mVertexBuffer* one by one. The whole Program 7 in fact can be replaced by simply one line: *mVertexBuffer.put(vertices.asReadOnlyBuffer())*, which means putting the entire *vertices* into *mVerteBuffer*. This change realizes the same functionality using the already existing library function, which is one of the key library functions already compiled into native code.

**Program 7** The **original** code of *CCGrid3D.blit().for_1*

```
for (int i = 0; i < vertices.limit(); i=i+3) {
  mVertexBuffer.put(vertices.get(i));
   mVertexBuffer.put(vertices.get(i+1));
   mVertexBuffer.put(vertices.get(i+2));
}
```

**Program 8** The changed Program 7

```
int limit = vertices.limit(); //added
for (int i = 0; i < limit; i=i+24) { //changed
   mVertexBuffer.put(vertices.get(i));
   mVertexBuffer.put(vertices.get(i+1));
   mVertexBuffer.put(vertices.get(i+2));
               ...
               ...
   mVertexBuffer.put(vertices.get(i+23)); //added
}
```

### C. Evaluation

Figure 7 shows the accumulative effects of the code changes on energy consumption. Exceptionally, "*Full-Use LF*" does not take previous changes into account and means only replacing Program 7 with the built-in library function as stated above.

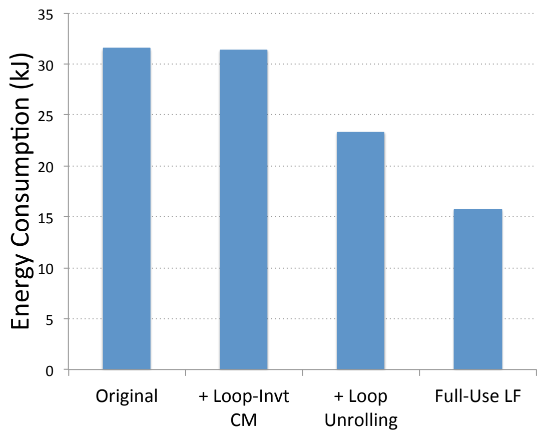


Fig. 7: Energy consumption of the code without and with the changes in `Orbit`.

We can see that loop-invariant code motion does not gain much energy saving because the *vertices.limit()* which is a library function as well uses a very small percentage of energy consumption. On the other hand, loop unrolling achieves 25.8% energy saving due to the reduction of amount of *Control Ops*, comparisons and assignments, which occupy most of the cost. And the most effective change is the replacement to a library function, saving 50.2% energy consumption because this library function has been complied into native code before execution, in contrast the java source code need runtime interpretation which is not free from energy consumption. The result indicates that it is a good idea for developers to make a good use of library functions rather than implementing the same function themselves with java source code.

## VIII. The `Waves` Scenario

In this section, similarly, we first analyze the energy characteristics of the blocks in the `Waves` scenario, based on which we modify the code and then evaluate the effects of changes on energy consumption.

### A. Energy Accounting

Unlike the `Orbit` scenario where only one block dominates energy consumption, in `Waves` scenario, the costs of top seven blocks are at the same order of magnitude of kJ, as listed in Table V. The *CCGrid3D.blit().for_1* is also employed in this scenario and is the most costly as well among all the blocks. The majority of blocks in Table V are directly or indirectly invoked by *CCWaves3D.update().for_1.for_1*, as shown in Program 9. And their jobs are mostly to set or get the values of member variables, so a large part of energy consumption goes to assignments and *Function Ops*. It was not expected that the code spends such a large amount of energy on simple setter and getter functions.

**Program 9** The **original** code in *CCWaves3D.update()*

```
int i, j;
for( i = 0; i < (gridSize.x+1); i++ ) {
   for( j = 0; j <(gridSize.y+1); j++ ) {
      CCVertex3D v = originalVertex(ccGridSize.ccg(i,j));
               ...
      setVertex(ccGridSize.ccg(i,j), v);
   }
}
```

**Program 10** Program 9 after Method Inline & Code Motion

```
ccGridSize ccgridsize = new ccGridSize(0,0); //added
CCGrid3D ccgrid3d = (CCGrid3D) target.getGrid(); //added
CCVertex3D v = new CCVertex3D(0,0,0); //added
int i, j;
for( i = 0; i < (gridSize.x+1); i++ ) {
   for( j = 0; j <(gridSize.y+1); j++ ) {
   ccgridsize.x=i;ccgridsize.y=j; //added
   v = ccgrid3d.originalVertex(ccgridsize); //changed
               ...
   ccgrid3d.setVertex(ccgridsize, v); //changed
   }
}
```

### B. Code Optimization

**Full-Use of Library Function:**
We have talked about the optimization for *CCGrid3D.blit().for_1* in Section VII-B where we replace the entire Program 7 with the one-line code, which makes use of library functions. We keep this change in this scenario. For other blocks, we come up with one modification as following.

**Method Inline & Code Motion:**
As shown in Program 9, the three functions called in the inner loop body are *CCGrid3DAction.originalVertex()*, *ccGridSize.ccg()* and *CCGrid3DAction.setVertex()*, which respectively cost 2891.3 Joule, 3769.1 Joule and 3285.4 Joule "In Application". Note that, *CCGrid3DAction* is the parent class of *CCWaves3D*, so *originalVertext()* and *setVertex()* can be directly called without referring to their class names. As seen in Program 10, we unpack these three methods in this block: the first and fourth "added" lines are unpacked *ccGridSize.ccg()*; the second "added" and first "changed" lines are unpacked *CCGrid3DAction.originalVertex()*; the second "added" and second "changed" lines are unpacked *CCGrid3DAction.setVertex()*. This change removes all the *Method Invocations*, parameter passing and value returns related to these three functions invoked by this block. Note that, the first three "added" lines are located outside the loop in order to reduce energy consumption of the process of initializing objects and calling *CCNode.getGrid()*.

TABLE V: In `Waves` scenario, the top 10 costly blocks "In Application". And the energy percentages of different kinds of operations in each block.

| Block ID | #Executions | Energy Cost (J) | Assi. | Decl. | Cont. | Func. | Bool. | Arit. | Libr. |
|---|---|---|---|---|---|---|---|---|---|
| CCGrid3D.blit().for_1 | 112193 | 8094.1 | 16.7% | 0% | 35.6% | 0% | 20.5% | 14.0% | 13.3% |
| CCVertex3D.CCVertex3D() | 40219 | 5232.0 | 27.2% | 0% | 10.0% | 62.8% | 0% | 0% | 0% |
| CCWaves3D.update().for_1.for_1 | 34604 | 4088.7 | 10.7% | 0% | 32.1% | 0% | 14.7% | 39.0% | 2.2% |
| ccGridSize.ccg() | 42275 | 3769.1 | 0% | 0% | 32.1% | 67.9% | 0% | 0% | 0% |
| CCGrid3DAction.setVertex() | 31856 | 3285.4 | 14.6% | 7.8% | 30.9% | 46.7% | 0% | 0% | 0% |
| CCGrid3DAction.originalVertex() | 36566 | 2891.3 | 19.1% | 10.2% | 40.3% | 30.4% | 0% | 0% | 0% |
| CCNode.getGrid() | 49119 | 2145.1 | 0% | 0% | 58.1% | 41.9% | 0% | 0% | 0% |
| ccGridSize.ccGridSize() | 10570 | 1173.8 | 30.3% | 0% | 31.6% | 38.0% | 0% | 0% | 0% |
| CCGrid3D.setVertex() | 3944 | 657.2 | 10.1% | 1.6% | 32.8% | 28.9% | 0% | 26.4% | 0.2% |
| CCGrid3D.originalVertex() | 2785 | 374.2 | 14.0% | 1.9% | 33.4% | 17.9% | 0% | 32.8% | 0% |

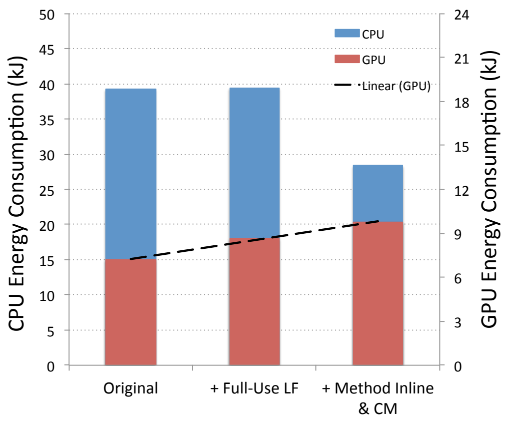


Fig. 8: CPU and GPU Energy consumption of the code without and with the changes in `Waves`.

### C. Evaluation

Figure 8 shows the accumulative effects of changes on energy consumption of CPU and GPU (note that previous figures only showed the CPU energy consumption because the GPU energy consumption did not vary noticeably before), and the dashed line indicates the linear trend of the GPU energy consumption. In the case of game, usually the aimed frames per second (FPS) is 60 Hz, when the game overloads CPU, the FPS will decrease, and when the workload is light, even very light, the FPS is generally fixed to 60Hz. The FPS in "*Original*" is around 36Hz; that in "+ *Full-use LF*" is around 50Hz; that in "+ *Method Inline & CM*" is around 60Hz. The change of *Full-Use LF* (full use of library function) does not save energy for CPU since the execution of original `Waves` actually overloads the CPU capacity, so the improvement of code enhances the performance and enables the device to generate more frames every second. Consequently, the GPU does more work and consumes more energy, as seen in Figure 8. After this change, when we apply the method inline and code motion, 27.7% of the overall CPU energy is saved, and for the same reason GPU consumes slightly more. This experimental result shows that our approach not only saves energy but also potentially boosts performance, which benefits users doubly.

## IX. Related Work

**Energy Modeling:**

From the hardware side, initial efforts on energy modeling research have been put on circuits-level (see the survey [28]), gate-level [26], [27] and register-transfer-level [15]. Later, research focus shifted towards high-level modelings, such as software and behavioral levels [25].

Energy modeling techniques for software start with the basic instruction level, which calculates the sum of energy consumption of basic instructions and transition overheads [8], [42]. Gang et al. [36] base the model at the function-level while considering the effects of cache misses and pipeline stalls on functions. T. K. Tan et al. [41] utilize regression analysis for high-level software energy modeling.

However, the run-time context considered in the above works is unsophisticated, free from user inputs, a virtual machine, dynamic compilation and so on. Furthermore the software stack below the level that they deal with (such as the level of the basic or assembly instruction) is relatively thin.

When research is focused on the energy of mobile applications, the level of granularity of the techniques is increased as well. An important part of such efforts is the use of operating system and hardware features as predictors to estimate the energy consumption at the component, virtual machine and application level [11], [18], [33], [37], [45], [46].

Shuai et al. [14] and Ding et al. [20] propose approaches to get source line energy information. The former requires the specific energy profile of the target system, and the workload is fine-tuned. The latter utilizes advanced measurement techniques to obtain the source line energy cost.

Compared with approaches above, Li et al. [21] explore the idea of identifying energy operations and constructing a fine-grained model based on operations which is able to capture energy information at a level more fine-grained than source line.

**Energy-Saving Techniques:**

A large amount of research efforts on energy-saving for mobile devices have been put on the main hardware components, such as the CPU, display and network interface. The CPU low-power-design techniques involve dynamic voltage-frequency scaling [22] and heterogeneous architecture [13], [23]. Techniques for display contain dynamically dimming the back-light [9], [32], tone-mapping based back light scaling [5], [16]. The network-related techniques try to exploit idle and deep sleep opportunities [24], [39], shape the traffic [10], [35] and so on.

There are many pieces of work relevant to code refactor for energy-saving . Vetro' et al. [43] define the concept of energy code smells that are the code patterns (such as self assignment, repeated conditionals and useless control flow) maybe energy-consuming. However, the code patterns selected in [43] have very little impact (less than 1.0%) on energy consumption. Our experimental result shows that our approach is able to save half of the entire energy consumption in certain scenario.

Ding et al. [19] conducted a small scale evaluation of several commonly suggested programming practices that may reduce energy. Its result shows that reading array length, accessing class field and method invocation all cost noticeable energy. However, this work only provides a small number of suggestions to developers on how to make the code more energy-efficient.

Compared with previous work, our research propose a systematic energy-aware programming approach, which is guided by the operation-based source-code-level energy model. The experimental result shows that this approach is an effective guideline for energy-aware mobile application development.

## X. Conclusion

In this paper, we propose an energy-aware programming approach for mobile app development, which is guided by the operation-based source-code-level energy model. The general steps of the approach are as following: 1) we construct the operation-based energy model by mining the data generated in a range of well-designed execution cases; 2) based on the model, we capture the energy characteristics of the code; 3) we improve the code by removing, reducing or replacing the expensive operations in the costly blocks.

We evaluate this approach on a real-world game engine and on a physical Android development board with two ARM quad-core CPUs. The experimental result shows that our approach has a significantly positive impact on energy-saving. For different scenarios, this approach can save energy by from 6.4% to 50.2%. The result also indicates that the performance of code is a byproduct as well of this approach, which potentially improves user experience more.

## References

[1] *Android Debug Bridge*. http://developer.android.com/tools/help/adb.html.

[2] *Cocos2d-Android*. https://code.google.com/p/cocos2d-android/.

[3] *Dalvik Virtual Machine*. http://source.android.com/devices/tech/dalvik/.

[4] *Report: U.S. Smartphone Penetration Now At 75 Percent*. http://marketingland.com/report-us-smartphone-penetration-now-75-percent-117746, 2015.

[5] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 57–70, New York, NY, USA, 2011. ACM.

[6] Android. *A JIT Compiler for Android's Dalvik VM*. http://www.android-app-developer.co.uk/android-app-development-docs/android-jit-compiler-androids-dalvik-vm.pdf.

[7] D. Bogdanas and G. Roşu. K-java: A complete semantics of java. *SIGPLAN Not.*, 50(1):445–456, Jan. 2015.

[8] C. Brandolese, W. Fomacian, F. Salice, and D. Sciuto. An instruction-level functionality-based energy estimation model for 32-bits microprocessors. In *Design Automation Conference, 2000. Proceedings 2000*, pages 346–350, 2000.

[9] W.-C. Cheng and M. Pedram. Power minimization in a backlit tftlcd display by concurrent brightness and contrast scaling. *Consumer Electronics, IEEE Transactions on*, 50(1):25–32, Feb 2004.

[10] C. Chiasserini and R. Rao. Improving battery performance by using traffic shaping techniques. *Selected Areas in Communications, IEEE Journal on*, 19(7):1385–1394, Jul 2001.

[11] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.

[12] C. Edwards. Lack of software support marks the low power scorecard at dac. In *Electronics Weekly.*, pages 15–21, June 2011.

[13] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 31(2):86–95, March 2011.

[14] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 92–101, Piscataway, NJ, USA, 2013. IEEE Press.

[15] C.-T. Hsieh, Q. Wu, C.-S. Ding, and M. Pedram. Statistical sampling and regression analysis for rt-level power evaluation. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 583–588, Nov 1996.

[16] A. Iranli and M. Pedram. Dtm: Dynamic tone mapping for backlight scaling. In *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, pages 612–617, New York, NY, USA, 2005. ACM.

[17] X. Jiang, P. Dutta, D. Culler, and I. Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 186–195, New York, NY, USA, 2007. ACM.

[18] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 39–50, New York, NY, USA, 2010. ACM.

[19] D. Li and W. G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, pages 46–53, New York, NY, USA, 2014. ACM.

[20] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 78–89, New York, NY, USA, 2013. ACM.

[21] X. Li and J. P. Gallagher. A top-to-bottom view: Energy analysis for mobile application source code. *CoRR*, abs/1510.04165, 2015.

[22] X. Li, G. Yan, Y. Han, and X. Li. Smartcap: User experience-oriented power adaptation for smartphone's application processor. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 57–60, San Jose, CA, USA, 2013. EDA Consortium.

[23] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. *SIGPLAN Not.*, 47(4):13–24, Mar. 2012.

[24] J. Liu and L. Zhong. Micro power management of active 802.11 interfaces. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 146–159, New York, NY, USA, 2008. ACM.

[25] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(11):1061–1079, Nov 1998.

[26] R. Marculescu, D. Marculescu, and M. Pedram. Adaptive models for input data compaction for power simulators. In *Design Automation Conference, 1997. Proceedings of the ASP-DAC '97 Asia and South Pacific*, pages 391–396, Jan 1997.

[27] F. Najm. Transition density: a new measure of activity in digital circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 12(2):310–323, Feb 1993.

[28] F. Najm. A survey of power estimation techniques in vlsi circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):446–455, Dec 1994.

[29] A. Ng. *CS229 lecture notes*. http://cs229.stanford.edu/notes/cs229-notes1.pdf, 2012.

[30] Odroid. *Odroid-XUE*. http://www.hardkernel.com/main/main.php.

[31] J. Pallister, S. Kerrison, J. Morse, and K. Eder. Data dependent energy modelling: A worst case perspective. *CoRR*, abs/1505.03374, 2015.

[32] S. Pasricha, M. Luthra, S. Mohapatra, N. Dutt, and N. Venkatasubramanian. Dynamic backlight adaptation for low-power handheld devices. *IEEE Design Test of Computers*, 21(5):398–405, 2004.

[33] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[34] G. Pinto, F. Castor, and Y. D. Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 22–31, New York, NY, USA, 2014. ACM.

[35] C. Poellabauer and K. Schwan. Energy-aware traffic shaping for wireless real-time applications. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 48–55, May 2004.

[36] G. Qu, N. Kawabe, K. Usarni, and M. Potkonjak. Function-level power estimation methodology for microprocessors. In *Design Automation Conference, 2000. Proceedings 2000*, pages 810–813, 2000.

[37] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 168–178, New York, NY, USA, 2009. ACM.

[38] Soot. *A framework for analyzing and transforming Java and Android Applications*. http://sable.github.io/soot/.

[39] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 261–274, New York, NY, USA, 2005. ACM.

[40] STACKOVERFLOW. http://stackoverflow.com.

[41] T. Tan, A. Raghunathan, G. Lakshminarayana, and N. Jha. High-level software energy macro-modeling. In *Design Automation Conference, 2001. Proceedings*, pages 605–610, 2001.

[42] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, Dec 1994.

[43] P. G. M. M. Vetro' A., Ardito L. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In *ENERGY 2013 : The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 34–39, March 2013.

[44] T. Šimunić, L. Benini, G. De Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings of the 13th International Symposium on System Synthesis*, ISSS '00, pages 193–198, Washington, DC, USA, 2000. IEEE Computer Society.

[45] C. Wang, F. Yan, Y. Guo, and X. Chen. Power estimation for mobile applications with profile-driven battery traces. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pages 120–125, Piscataway, NJ, USA, 2013. IEEE Press.

[46] L. Zhang, B. Tiwana, R. Dick, Z. Qian, Z. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 105–114, Oct 2010.