OPEN ACCESS

UNIVERSITY OF BRISTOL

Morse, J. C. M., Kerrison, S., & Eder, K. I. (2016). On the infeasibility of analysing worst-case dynamic energy. Manuscript submitted for publication. http://arxiv.org/abs/1603.02580v1

Early version, also known as pre-print

Link to publication record in Explore Bristol Research
PDF-document

University of Bristol - Explore Bristol Research

General rights

On the infeasibility of analysing worst-case dynamic energy

Jeremy Morse, Steve Kerrison and Kerstin Eder
University of Bristol

March 9, 2016

Abstract

In this paper we study the sources of dynamic energy during the execution of software on microprocessors suited for the Internet of Things (IoT) domain. Estimating the energy consumed by executing software is typically achieved by determining the most costly path through the program according to some energy model of the processor. Few models, however, adequately tackle the matter of dynamic energy caused by operand data. We find that the contribution of operand data to overall energy can be significant, prove that finding the worst-case input data is NP-hard, and further, that it cannot be estimated to any useful factor. Our work shows that accurate worst-case analysis of data dependent energy is infeasible, and that other techniques for energy estimation should be considered.

1 Introduction

A significant design constraint in the development of embedded systems is that of resource consumption. Software executing on such systems typically has very limited memory and computing power available, and yet must meet the requirements of the system. To aid the design process, analysis tools such as profilers or maximum-stack-depth estimators provide the developer with information allowing them to refine their designs and satisfy constraints.

A less well studied constraint is the limited energy budgets that deeply embedded systems possess. A typical example would be a wireless sensor powered by battery, that must operate for a minimum period without the battery being replaced. Other examples would be systems dependent on energy harvesting, or systems with low thermal design points that thus have a maximum power dissipation level. These constraints can also be approached with software analysis tools, and several techniques have been developed that allow the estimation of software's energy consumption [17, 7, 18].

Within energy estimation, focus has been given to *Worst Case* Energy Consumption (WCEC): determining the maximum amount of energy that can be consumed during the execution of the software. In this paper, we shall study the calculation of worst case energy, considering only the effects that different software and inputs can have on a system. The objective is to determine whether it is possible to establish an upper bound on energy that is tighter than over-estimating by, for example, using a maximum activity factor. Such a factor may be unachievable during the execution of a real program, because data that triggers the highest energy consumption in one instruction may, through data dependency and other constraints, preclude subsequent instructions from consuming their maximal energy [11].

1

Energy is the integral of power over a given period. The power dissipation of a processor can be apportioned in two parts: static and dynamic. *Static power* or leakage is the power dissipated for as long as the component is turned on, irrespective of its internal state or any changing inputs and outputs. *Dynamic power* or switching activity refers to power dissipation from the substance of execution: the switching of gates and charging of data buses, which all consume energy. We express these more formally in Section 3. Analysis of worst-case instantaneous dynamic power has been well studied in the literature, but here we consider worst-case energy, i.e. the power over a program execution.

Estimating worst case energy for a particular program naturally becomes the computation of these two distinct sources of energy consumption. Static power is directly controlled by the length of the program, measured in time. Numerous techniques have been developed by the *worst case execution time* (WCET) community to address this matter [33]. Dynamic power, however, has received much less attention. Several models of how systems consume energy have characterised the dynamic power only for specific inputs, averaged over all inputs, assumed a worst-case dynamic power for each instruction with few details, or assumed no dynamic power at all [12, 7, 32].

This paper demonstrates that for data-dependent dynamic energy, the calculation of the worst case input to a software execution is an NP-hard problem, and further, that its dynamic energy cannot be approximated to a useful factor. Our proof applies to processors in general, but we show on an example processor, the Xcore XS1-L [19], that the portion of energy that is infeasible to analyse contributes at least half of the processor's dynamic power.

The rest of the paper is structured as follows: in Section 2 we examine the current state of energy estimation, and related work. In Section 3 we study the Xcore processor and its dynamic energy consumption. Section 4 formalises the problem that we are dealing with, which is shown to be NP-hard in Section 5, and in Section 6 we demonstrate that the problem cannot be effectively approximated. We discuss the results in Section 7, and draw conclusions in Section 8 with an outlook on future work.

2 Background

This section identifies existing techniques for determining the energy consumption of software when executed, techniques for determining the maximum amount of energy a program can consume, and the theoretical definition of the MAXSAT problem.

2.1 Energy estimation techniques

Given the high complexity of microprocessors, energy analysis based on hardware designs tends to be resource intensive, and require access to proprietary design materials. Research has instead focused on using empirical techniques to model how processors consume energy. These models can then be used to estimate the consumption of a real-world system.

One of the most popular techniques is the instruction level energy model [30]. Various test patterns of instructions are executed on a processor and their power empirically measured, leading to a model of per instruction energy costs and the dynamic cost of switching between different instructions. Simulating an instruction sequence, or interpreting a trace of an execution, can then be combined with this energy model to produce a cost value for the execution. Steinke [27] extend this model to include the costs of circuit switching in instruction operands. These costs include the amount of switching occurring on data buses supplying input operands when executing instructions, and the switching on the output datapath when an operand or memory address is written to by an instruction.

Further modelling techniques for dynamic power go beyond the core part of the processor, such as analysing flash memory [21], caches [4] and DRAMs [16]. High performance processors feature hardware-provided counters that record metrics such as cache hit rates, which can be used by appropriately parametrised energy models [24]. In this paper, where embedded devices are the focus, we choose to only examine the dynamic power attributable to the core part of the processor.

2.2 Worst case energy consumption (WCEC)

WCEC is a form of energy estimate, where the aim is to find the maximum amount of energy that a piece of software will consume. The problem is thus made of two parts: modelling the energy consumption of the software under test, and searching for the execution of it that will lead to the greatest amount of energy consumed. This problem is similar to the worst case execution time problem (WCET) [33] where the execution time of software is modelled, and then the longest possible path found. For both problems, a *specific* worst case execution is sought. However, much interest is also shown in providing an upper bound on the worst case. Such a bound may be higher than the worst case, but may help demonstrate that a design constraint is met.

2.2.1 WCEC is not a simple reinterpretation of WCET

The worst case *energy* consumption problem goes beyond the worst case execution time problem, because the execution time of a single instruction is largely independent of its input data. This is because timing variability has mostly been eliminated "by design" through the use of synchronous logic and the limited propagation time associated with executing individual instructions.

In real-time embedded systems, timing-predictable processors execute instructions within a fixed number of clock cycles, irrespective of the data the operation works on. This is particularly beneficial to WCET analysis, which can then focus on identifying the worst case execution path which is determined by the control flow, rather than by the data flow of the computation. More advanced micro-architectural features, such as early-out of operations, or cache hierarchies, provide higher average performance at the cost of predictability. This makes WCET analysis far more challenging, as tight bounds firmly rely on timing predictability of the target architecture [28]. However, even operations that have a variable execution time can be quantized by the processor's clock period into a tractable number of discrete possibilities. The range may be in the order of tens, hundreds, or thousands of cycles, depending on the type of operation.

Energy depends on both, the execution time and the power dissipation of the operation. Power is not quantized in terms of the clock period, but instead by the number of transistor and interconnect state changes (i.e. switches) that *may* take place during an operation, depending on the data to be processed. The number of possible power dissipation levels is thus in the order of the number of transistors in the device. This is several orders of magnitude larger than the number of timing possibilities explored by WCET analysis.

For the techniques that are used in WCET to be directly transferable to WCEC, a set amount of energy per operation would need to be specified and realised in hardware, similar to specifying and ensuring, through timing analysis, that each operation fits into a fixed number of clock cycles. Consider the converse: A processor that presents a similar WCET analysis difficulty to determining WCEC, would be an asynchronous design, where the precise execution time is a non-trivial function of an operation's input data. Such devices may have an average delay, but actual performance for a given use case or tight bounds may be harder to determine [14].

2.2.2 Existing work on WCEC

The first publication to provide a technique for computing the WCEC of software was [12], where upper bounds on the energy consumption of several programs were inferred using energy models of software basic blocks and an ILP solver to find a maximal path through the program. The authors additionally debunk the suggestion that the execution path consuming the most time is always the path that also consumes the most energy. With regards to dynamic power, the authors assume that all circuits switch on every clock cycle rather than attempting to determine actual switching activity, their justification being that the contribution of dynamic power to overall energy is low, thus their approximation does not introduce significant imprecision. We address this in Section 3.

Resource analysis techniques that extract cost relations from programs have been employed to analyse energy consumption bounds [17, 7]. The costs used in these analyses represent energy consumption and are based on models that provide a single energy cost per instruction, obtained by averaging the energy measured from processing random data, constrained to yield valid operands for the respective instruction [15]. However, bounds obtained in this way cannot be considered safe, as executions would exist where the energy from operand data exceeds the average case.

More recently, [32] have presented techniques for estimating over and under approximations of WCEC through implicit path enumeration and genetic algorithms, respectively. They do not, however, comment on dynamic power at all: their absolute instruction energy model appears to assume maximum switching for each instruction cost. Their relative energy model does not consider real energy costs, instead estimating the difference in energy consumption between instructions, again with no explicit consideration of dynamic power.

Both Jayaseelan and Wägemann identify inefficiency as being a reason why they cannot compute accurate switching activities for circuits. As we will show in this paper, the problem is infeasibly complex under the $P \neq NP$ assumption.

2.3 Existing complexity results

Switching activity is a matter studied in detail by the VLSI community for circuit design, as the maximum instantaneous switching in a circuit can affect the power supply requirements [20]. This problem has been shown to be NP-hard [5] and numerous techniques have been developed to make an estimate of the worst case power consumption [8], allowing maximum power analysis.

Power estimation itself does not directly correspond with energy estimation. The objective of WCEC is finding the maximal amount of circuit activity over a period of time, rather than the instantaneous maximum, which itself may be incompatible with the circumstances that lead to maximum energy. In particular, software requires that computations be consistent with past inputs, creating additional constraints and dependencies.

Switching between instructions is a notable source of energy consumption, which can be controlled through the order in which instructions are executed. Techniques have been developed to reduce consumption through instruction scheduling [23], but this is known to be an NP-hard problem. Instruction scheduling uses pre-computed costs of switching between instructions to determine an optimal static schedule. It does not consider the data operands to instructions or any cost that does not have a fixed value.

None of these complexity results are directly applicable to the estimation of energy in data-dependent switching during software execution. To the best of our knowledge, we believe this is the first work to consider data-dependent switching costs.

2.4 Maximum satisfiability

The *Maximum satisfiability* problem "MAXSAT" [2, pp.613–631] is defined as the satisfiability problem where the number of clauses satisfied must be maximised by an assignment. Following the presentation of [13], define L to be a set of literals, and C a set of disjunctive form clauses:

$$L = \bigcup_{i>0} \{x_i, \overline{x}_i\}$$

$$c \in C, C = \{l_1 \vee ... \vee l_n \mid l_i \in L\},$$

where each x_i is a Boolean variable. A truth assignment defines each x_i or its negation to be true. A clause is deemed to be satisfied if at least one literal in the clause is assigned true. A MAXSAT problem is a set of literals and set of clauses $\langle L, C \rangle$, such that the solution is the truth assignment that causes the maximal number of clauses to be satisfied.

3 Circuit switching on Xcore

Prior WCEC papers have relied on the suggestion that the variation in dynamic switching is small in relation to other energy costs in a processor, at approximately $3\,\%$ [29]. Other work has presented a mixed picture: [26] found that the switched capacitance (i.e. switching cost) of a StrongARM processor had little variance across applications, suggesting that switching costs contribute little to overall program energy; while [1] observe that data switching accounts for up to 50% of processor core energy.

Here, we affirm that dynamic switching costs can be high by analysing the energy consumption of the Xcore [19] XS1-L, and demonstrating a significant energy variation due to dynamic switching.

3.1 Defining power dissipation in a micro-processor

The energy, E, of an electronic device is the integral of its power dissipation, P, over a given time period, T:

$$E = \int_{t=0}^{T} P(t) \; dt. \tag{1}$$

Power is an instantaneous measure of the rate of work. Typically, this is sampled repeatedly in order to discretise the integral, or the power is averaged, simplifying the equation to $E = P \times T$.

In digital devices such as processors, the total power dissipation of the device, P_{tot} is typically apportioned into two additive parts, termed static and dynamic, denoted here as P_s and P_d respectively:

$$P_{tot} = P_s + P_d \tag{2}$$

Elaborating on these, static power is determined by the operating voltage, V_{dd} of the device and I_{leak}, the leakage current present, which is itself dependent upon physical characteristics such as operating temperature, transistor feature size and the manufacturing process that is used.

$$P_s = V_{dd} I_{leak}, \quad \therefore \quad P_s \propto V_{dd} \tag{3}$$

Dynamic power is dependent upon the capacitance of the components that are being switched, C_{sw}, as well as the operating voltage and the frequency of switching, f. In a processor, f is

governed by the clock frequency. The proportion of the device that is switching is dependent upon the instruction and data being executed and related changes in state. This is represented by an *activity factor*, α, where each instruction or action performed by the processor may have a different α.

$$P_d = \alpha C_{sw} V_{dd}^2 f, \quad \therefore \quad P_d \propto V_{dd}^2 \tag{4}$$

There is a quadratic relationship between voltage and dynamic power. The necessary operating voltage is approximately linearly proportional to the operating frequency.

3.2 Apportioning dynamic power

When considering power per instruction, it is important to calculate an appropriate α per instruction, or some equivalent by abstraction. However, the instruction is not the sole influence upon the α value. The operands supplied to the processor's functional units (for example, arithmetic unit), will affect the amount of switching. This includes changes to the input and output data buses, as well as internal switching within the functional unit as the new result is computed. As such, one instruction may have a range of possible α values that are dependent on the input data.

Prior work [12] has suggested that this variation in α is small and therefore not significant enough to consider when constructing a worst-case energy model. However, we demonstrate that variation in input data can be responsible for as much as 42 % of a core's power dissipation and thus becomes a relevant contributor to the model. This is pertinent to systems with minimal additional components, such as those that are deeply embedded, where the processor is the major consumer of energy. In larger, more complex systems, with many external devices and power supplies, the variation in total system energy due to data values is proportionally smaller.

Internal processor data buses are one of the largest contributors of dynamic power. These buses interconnect various internal units, and so changing values on these buses indicate the charge or discharge of connections between a number of gate inputs and outputs, which may have different loads depending on their fan-in or fan-out and connection length. The [27] energy model explores this and discovers that approximately 20 % of overall processor power can be attributed to the Hamming distance on buses.

To determine the dynamic power cost on our target device (the Xcore XS1-L), we performed experiments in the manner of [27]. For a set of instructions, we tested every combination of operand inputs from zero to 255 for each operand, creating a sequence of tests, $\mathbb{P}$. We alternate between instructions with this data set and all-zero operands, to ensure we measured the Hamming distance on each cycle. The Xcore is a cache-less multi-threaded processor with time-deterministic execution. Test sequences were constructed in such a way to ensure we exercised the processor datapaths in every instruction cycle. Although the processor has a 32-bit datapath, exhaustive testing over 8-bit data is sufficient to expose the behaviours of interest to this work.

The device is operated with a 1.0 V core power supply and 500 MHz clock frequency. Power is sampled at the 3.3 V input to the DC-DC converter that supplies the cores and is done so using a vendor-supplied sampling and debug device that uses a shunt resistor to determine current. The tests are each run repeatedly for a 0.5 s duration in order to acquire several thousand power samples, then taking the average.

The device under test is a dual-core component, tested with single-core code. As such, we must remove the additional energy consumption that would not be present if a single-core version of the component were to be used. This is established through the following steps. First, measure the power of the dual-core device when idle, defining P_{tdual} to ascertain $P_{tsingle}$:

$$P_{tsingle} = \frac{P_{tdual}}{2}. \tag{5}$$

Executing instruction power tests on one core, leaving the remaining core idle, produces a sequence of test results, $\mathbb{P} = \{P_0, \ldots, P_n\}$. Define the dynamic power contribution of the lowest and highest power test cases as P_{dmin} and P_{dmax} respectively, and the dynamic power range, P_{drng}:

$$P_{dmin} = \min(\mathbb{P}) - P_{tdual}, \tag{6}$$

$$P_{dmax} = \max(\mathbb{P}) - P_{tdual}, \tag{7}$$

$$P_{drng} = P_{dmax} - P_{dmin}. \tag{8}$$

We observe for the device under test that $P_{tdual} = 328\,\text{mW}$ and therefore $P_{tsingle} = 164\,\text{mW}$. Any additional power observed during tests is used to determine how much dynamic power variation is possible for the set of input values tested. This is not solely static power, because even at idle, switching in components such as the clock tree is taking place, contributing to dynamic power. Thus, the difference in power observed during instruction and data tests is not the total dynamic power contribution, but does establish the degree of variation in dynamic power that can take place, and what proportion of total core power this amounts to.

For the add instruction, $P_{dmin} = 34\,\text{mW}$ and $P_{dmax} = 96\,\text{mW}$, giving $P_{drng} = 62\,\text{mW}$. This demonstrates that for add, up to 27 % of the core's power dissipation is governed by operand values. In a system where processor power is significant, this is a substantial variation, inaccurate predictions of which may be undesirable.

To aid analysis of the results of these experiments, we present a series of "heat-map" figures, showing measured dynamic power in colour and datapath Hamming weights in greyscale. These plots use measurements from tests of the add instruction.

Figure 1 shows total dynamic power for add with all combinations of two 8-bit operands. The diagonal striping indicates a strong correlation with the number of bits set to 1 in the result of the computation. This is observable due to alternating between test add operations and all-zero operations. The Hamming weight of the output is shown in Figure 2. This is determined to represent 4.4 mW per output bit set.

Subtracting the calculated switching power per output bit from the original dynamic power measurements gives Figure 3. This reveals a second pattern that was previously obscured by the dominant effects of the output Hamming weight. Intuitively, this corresponds to the Hamming weight of both input operands, demonstrated in Figure 4. We determine this to be 1.3 mW per input bit set. Repeating this process and subtracting the calculated power per input bit gives Figure 5, which closely corresponds to the Hamming weight shown in Figure 2 as previously stated.

Finally, by subtracting both of the input and output bit dynamic powers produces Figure 6, which shows that the remaining variation in dynamic power is an order of magnitude lower than the effect of these Hamming weights, ranging from 12 mW to 0 mW. Expressed as a series of matrix operations, where P is the measured dynamic power and the input and output Hamming weights are presented as H_i and H_o respectively, the remaining unaccounted for dynamic power, D of Figure 6, is:

$$D = P - (H_i \cdot 1.3) - (H_o \cdot 4.4) \quad \text{mW} \tag{9}$$

In a real-world program, switching between an operation and all zeroes would not take place. However, with each new instruction, a Hamming distance would be present between the previous and current input values, as well as the previous and current output value. As such, the properties described here naturally translate from Hamming weights into Hamming distances.
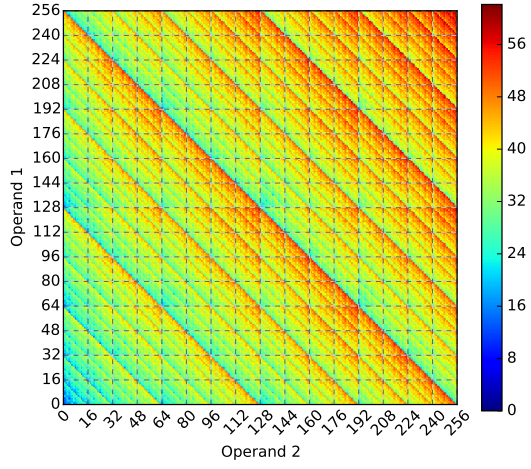
7

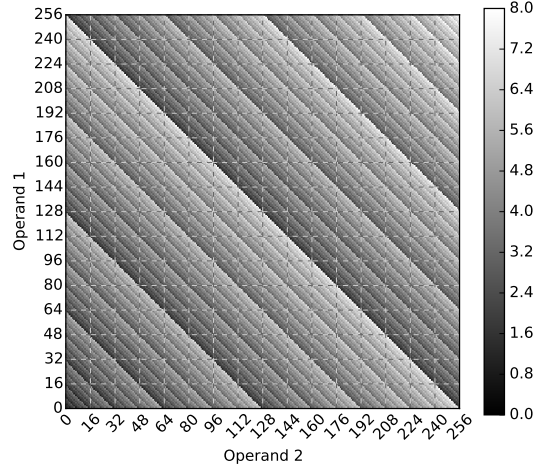Figure 1: Dynamic power in milliwatts for add instruction.

Figure 2: Hamming weight of output datapath of an add instruction, in number of bits set.
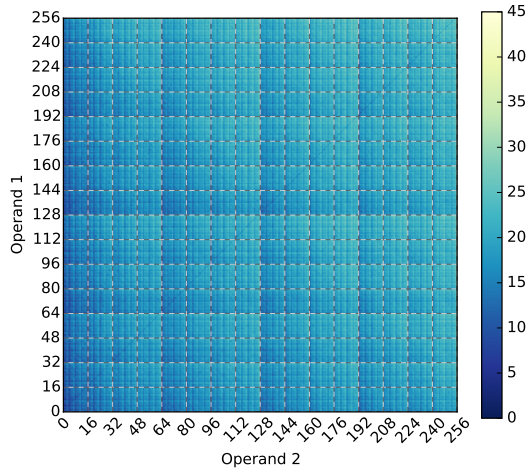

Figure 3: Dynamic power in milliwatts for add instruction, with output datapath cost subtracted (assuming 4.4mW per bit).
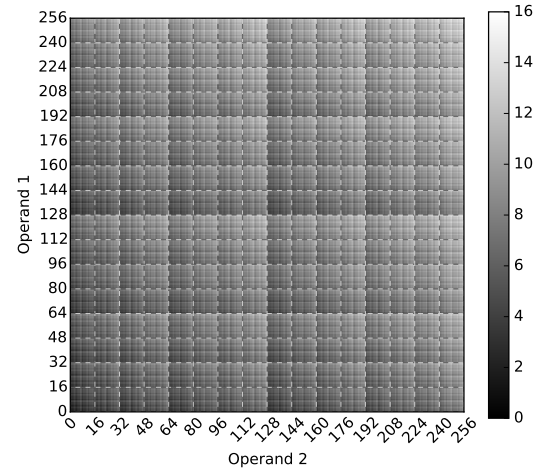

Figure 4: Hamming weight of both input operands to an add instruction, in number of bits set.
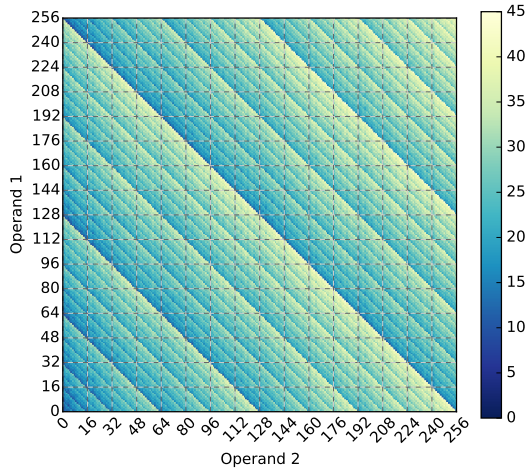
Figure 5: Dynamic power in milliwatts for add instruction, with input datapath cost subtracted (assuming 1.3mW per bit).
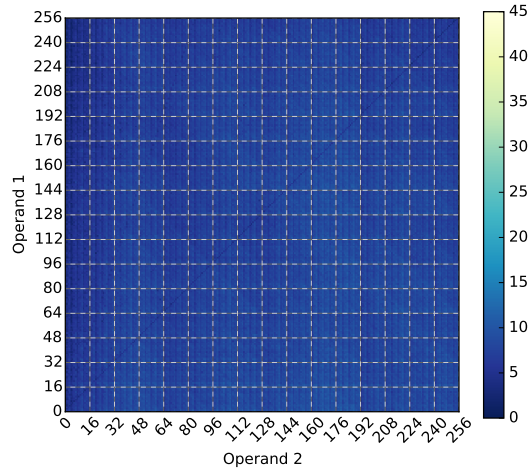

Figure 6: Dynamic power in milliwatts for add instruction, with output and input datapath costs subtracted.

Across all of our experiments, the maximum dynamic power observed was $123\,\mathrm{mW}$, caused by the `sub` instruction. This is due to `sub` producing a negative two's complement output that results in all bits being set in the output operand, causing maximal Hamming distance in the output datapath. However, compared to the base instruction cost this means that on the Xcore the dynamic switching contributes as much as $42\,\%$ of the total processor power. Similar work for 8 bit AVR [22] shows dynamic power making up $15\,\%$ of processor power.

This data demonstrates that, at least on the Xcore and the AVR, the contribution of dynamic power to the full processor cost is non-trivial, and certainly a significant contributor to calculating the worst case energy in a program. We also observe that the output datapath for our particular processor is the most significant contributor to dynamic power. Accordingly, and for simplicity, we focus only on this component in subsequent sections: specifically, the Hamming distance between values on the output datapath between subsequent cycles.

3.3 Summary and Discussion

With regard to prior work that analyses the significance of dynamic power in software execution, we have demonstrated that on the Xcore dynamic power can be a large proportion of overall energy consumption by the processor, but cannot discount prior work that found little contribution on other platforms. This suggests that dynamic power contribution *can* be significant, but that it varies from processor to processor.

The system context should also be considered, for two main reasons. Firstly, a system that features a display and backlight component will have its total energy consumption dominated by these over all other components [3]. Looking beyond embedded systems, large multi-core processors such as the Xeon Phi [25] consume significantly more energy in caches and memories than in computation. This will of course significantly reduce the impact of any variation in processor energy. Secondly, the type of system and its performance requirements will influence processor choice, and the amount of power variation of the chosen processor will determine whether it is necessary to consider it. If this is the case, the computational workload placed

upon the system will then determine how much each part of the processor is exercised. It is shown in [9] that both processor choice and workload change how processor subcomponents such as the register file and functional units contribute to total energy consumption.

With this in mind, we observe that consideration of dynamic power caused by data operands is most relevant for applications in the Internet of Things (IoT) domain. Such applications typically have energy budgets as a primary concern, have some non-trivial processing task that requires a microcontroller, but do not use a large processor featuring caches and other performance enhancing hardware. When attempting to meet design constraints such as battery lifetime, determining the worst case energy consumption of software would be of interest, and thus determining the impact of data operands on dynamic energy consumption.

4 Formalising the circuit switching problem

As illustrated in the previous section, the matter we consider is the amount of energy caused by circuit switching, specifically the switching occurring on the output datapath in a processor. Here, we formalise our problem, which we name the "Circuit SWitching Problem" (CSWP), discussing its limitations and generality. Our objective is to take a program, determine the maximum amount of output datapath switching activity that can occur in that program, and, in the process, find the program input that triggers it.

Because we are concerned with the amount of circuit switching that can occur, we choose to avoid any facility for varying the length of a program in this formalisation, i.e. the number of instructions executed. A CSWP program thus cannot have any branch instructions or conditional execution ability: it corresponds closely with a trace of a general program execution, or a general program that has been unrolled and all conditional branches eliminated. Dealing with programs of varying length would unnecessarily involve searching different paths through the program.

Formally, we consider a CSWP program, P, to be a finite sequence of n instructions, x_i, such that $P = x_1, x_2, ..., x_n$. Each instruction is a 3-tuple $\langle m, i, o \rangle$, where m is a mnemonic $m \in M$, i is a set of inputs (discussed below), and o is an output operand. Both input and output operands (discussed further below) are considered to be bit-vectors of width w.

A CSWP program executes on an abstract machine with a monotonically incrementing program counter, an infinite number of registers, and a memory store of finite size. Memory is considered to be an array of size 2^w with each memory cell a bit-vector of width w. For each instruction x_i in the CSWP program the machine takes the input operands, computes an output according to the function of the instruction mnemonic, and writes the result to the output operand. The objective function of CSWP is then to compute:

$$\sum_{i=1}^{n-1} h(o_i, o_{i+1}) , \tag{10}$$

where h is a function computing Hamming distance between two values, i.e. the output values of each subsequent instruction, corresponding to the output datapath of the abstract machine.

Each mnemonic $m \in M$ represents a function over the input operands, resulting in a single output. In line with the constraints detailed above, CSWP programs only perform arithmetic computations, mapping input operands to an output. There are no branch mnemonics, neither are there any instructions that induce side effects of any form (such as changing some state or the program counter). We do not define a set of mnemonics that a CSWP program may use, however for the purposes of this paper we write listings using standard RISC mnemonics such as add, sub, ldr, mov [10].

Each input operand is permitted to be one of four classes of sources:

- Free inputs, which we denote with the text *freebit*.

- Constant values, which we write in hexadecimal.

- A memory access to a fixed address $m[x]$, with x the address.

- The output operand of a prior instruction, written o_i, where for the current instruction x_j, $i < j$.

The value of every input is always a bit-vector of width w. Free inputs may take any value, likewise constants may only have one value, defined in the instruction being executed. In our examples below, we further assume that all free inputs only take the values zero or one. Memory accesses evaluate to the contents of a memory cell, but for simplicity we only permit the addressing of fixed memory addresses. Prior output operands correspond to the output of each instruction being written to one of the infinite registers, which may then be read as an input to another instruction.

All instructions are considered to have an output of bit-width w, i.e., they all write some value to the output datapath of the machine. A `nop` (no-operation) instruction would be any instruction that repeats the output value of the previous instruction, causing no switching activity on the output datapath. Outputs may optionally be written to a memory cell $m[x]$, where x is a fixed address for the output value to be written to. In this circumstance, the output value may still be referred to as o_i, as a store to memory still causes the bits in the machine's output datapath to flip.

This formalisation has a number of limitations, most notably that without an infinite data store or ability to programmatically address it, it is not Turing complete. Given that our aim is to find the maximum switching for a particular path through a general program, this is a suitable restriction. The formalisation does not correspond to a particular machine, although with additional restrictions it may correctly model the execution trace of existing processors. The memory array may be considered to be superfluous given the lack of complex addressing, however it provides a useful mechanism for illustrating our examples through the rest of this paper.

We observe that CSWP is in class NP, as one may easily check the validity of a solution. Given the CSWP program and an input valuation for each free input, we can simulate the program with the given inputs, counting the number of bit flips at the same time process. The complexity of this process scales linearly with the number of instructions, n.

5 Reducing MAXSAT2 to circuit switching problem

To demonstrate that the CSWP is NP-hard, we must reduce any NP-hard problem to CSWP in polynomial time. For this, we turn to the MAXSAT problem, which is known to be NP-hard [2]. Specifically, we work with the MAXSAT2 variant, where each clause is limited to having at most two literals. Despite 2SAT being solvable in polynomial time, MAXSAT2 is still known to be NP-hard [31].

We reduce MAXSAT2 to CSWP by simulating MAXSAT2 in the switching activity of an instruction sequence, where the input that causes the maximum amount of circuit switching corresonds to an assignment to the Boolean variables that causes the maximum number of clauses to be satisfied. The reduction is illustrated in Algorithm 1, which takes the number of Boolean variables and the set of clauses as input, and outputs a CSWP program that simulates MAXSAT2. Here, we assume that the function `PrintInsn` causes a CSWP instruction to be emitted from the algorithm, with the instruction mnemonic, set of variables, and optional output

destination as its respective arguments. The return value identifies the output operand of the instruction.

Algorithm 1: Algorithm for encoding of MAXSAT2 formula within a CSWP program, printed via PrintInsn.

Input: Number of variables n and set of clauses C
Output: CSWP program encoding MAXSAT2 problem
var_addr = 0;
for $i = 0$; $i < n$; $i{+}{+}$ **do**
    out1 = PrintInsn(*"mov"*, *[freebit]*);
    out2 = PrintInsn(*"xor"*, *[out1, 0x1]*);
    PrintInsn(*"store"*, *[out1], m[var_addr++]*);
    PrintInsn(*"store"*, *[out2], m[var_addr++]*);
    PrintInsn(*"mov"*, *[0]*);
end
foreach $c \in C$ **do**
    $< l1, l2 >= c$;
    laddr1 = LitToMemAddr(*l1*);
    laddr2 = LitToMemAddr(*l2*);
    lit1 = PrintInsn(*"load"*, *[m[laddr1]]*);
    PrintInsn(*"xor"*, *[lit1, 0x1]*);
    PrintInsn(*"mov"*, *[0]*);
    lit2 = PrintInsn(*"load"*, *[m[laddr2]]*);
    PrintInsn(*"xor"*, *[lit2, 0x1]*);
    PrintInsn(*"mov"*, *[0]*);
    PrintInsn(*"or"*, *[lit1, lit2]*);
    PrintInsn(*"mov"*, *[0]*);
end

First, we read a series of free input values that we assume lie in the range $[0, 1]$, i.e. represent true or false in the lowest bit of the bit-vector. We consider each of these bits to be an assignment to a Boolean variable in the MAXSAT2 problem. Each bit, and its compliment, are stored to a location in memory. This creates an array of values corresponding to the truth of each literal. At the end of this process we insert a `mov` instruction that loads a zero value, for the purpose of resetting the value on the output datapath to zero. The net effect is that for each Boolean variable read, a constant amount of switching activity occurs. Consider each value the free variable may have:

1. `True`: Reading the input switches the lowest output datapath bit to on, the subsequent `xor` switches it to off, and the final `mov` causes no switching.

2. `False`: Reading the input causes no switching, the `xor` switches the lowest output datapath bit to on, and the subsequent `mov` switches it back to off.

Thus, for each Boolean variable read, the CSWP program always causes two bit flips.

We then proceed to use the memory region prepared with literal valuations to simulate the MAXSAT2 problem. We assume a mapping between each literal of the Boolean variables and the address of its valuation in the memory array, and use the function `LitToMemAddr` to translate from literal to memory address. Then, for each clause, we produce an instruction sequence that loads each literal valuation using the constant-switching technique used to read free inputs. Once

the literals are loaded, they are `or`'d together, after which the output datapath is loaded with zero again.

The CSWP program produced by Algorithm 1 has both a constant and data dependent portion of switching activity. Two bit-flips occur for each Boolean variable in the input MAXSAT2 problem, and four for each clause. The switching activity from the `or` instruction, however, directly corresponds to the satisfiability of the clauses: if a clause is satisfiable (i.e., one of the literals is true) then the `or` and following `mov` will cause two additional bit-flips. If a clause is not satisfiable, the same instructions will cause no switching. As a result, the maximum amount of switching in the program is caused by the maximum number of clauses being satisfied. The assignment to the free variables which causes this is also an assignment to the Boolean variables of the MAXSAT2 problem that causes the maximum number of clauses to be satisfied. As a result, CSWP must be at least as hard as MAXSAT2 (i.e. NP-hard). As we know CSWP is also in class NP (Section 4), CWSP is NP-hard. □

We observe that the reduction is performed in polynomial time, as it scales linearly with the number of Boolean variables n and the number of clauses, of which there can be at most n^2.

Given this result, we can conclude that there cannot be an efficient algorithm that solves the CSWP, unless P = NP. Thus, given that general programs can be unrolled and reduced to a CSWP, it is infeasible to determine the worst case datapath switching in a program, defeating energy estimation techniques that would rely on such a model. However, given such a limitation, there could still be algorithms that approximate the worst case switching to a certain degree of accuracy, allowing worst case switching to be narrowed down to a small range of values. We address this in the next section.

6 Inapproximability

Algorithm 2: Algorithm encoding a SAT problem into CSWP, with an output gap governed by satisfiability

---

Input: Number of variables n and set of clauses C
Output: CSWP program with switching gap
```
/* Decision phase                                                          */
base_var_addr = var_addr = 0;
insn_count = 0;
for i = 0 ; i < n ; i++ do
    out1 = PrintInsn("mov", [freebit]);
    PrintInsn("store", [out1], m[var_addr++]]);
end
result = CheckSat(base_var_addr, C);
bit_pattern = PrintInsn("ite", [result, 0xFFFFFFFF, 0]);
/* Switching phase                                                         */
decision_insn_count = insn_count;
for i = 0 ; i < ((decision_insn_count)/2 + 1) ; i++ do
    PrintInsn("mov", [bit_pattern]);
    PrintInsn("mov", [0]);
end
```

Having shown that CSWP is NP-hard, we will now show that it also cannot be approximated to any useful factor. We demonstrate that there is no constant ε for which an approximation

factor of $1 - \varepsilon$ can be achieved, and then that polynomial approximation factors also cannot be achieved. Intuitively, this is because each bit flip caused by the program is the product of an arbitrary computation, meaning there is no structure to the combinatorial problem that one can generally rely upon when constructing an approximation.

Formally, we demonstrate CSWPs inapproximability using a gap introducing reduction [31] from SAT to CSWP. Such a reduction transforms an NP-complete decision problem into an NP-hard optimisation problem, with a quantity (the "gap") of the feature being optimised governed by the truth of the decision problem. By demonstrating such a gap, one shows that a portion of the NP-hard problem cannot be approximated in polynomial time, as the approximation algorithm would have to solve a NP-complete problem in the process.

In the context of CSWP, we demonstrate that for any instance of SAT, problem p, we can reduce it to a CSWP program q where a portion of the switching activity is governed by the truth of whether p is satisfiable. The transformation is illustrated in Algorithm 2, which we divide into two discrete portions: the decision phase, and the switching phase. We use the same functions as in Algorithm 1, with the modification that the `PrintInsn` function increments a counter, `insn_count`, for every instruction printed.

Throughout the decision phase, we are not concerned with the switching activity that may occur, and do not seek to control it, in contrast with the previous algorithm. We begin by reading n free variables, which we assume to be bit-vectors valued either zero or one, and store them to fixed addresses in memory. We then pass the address of the variable valuations and the SAT clauses to the `CheckSat` function, which emits a CSWP program that evaluates the clauses over the Boolean variables stored at `base_var_addr`, and returns an output operand identifying whether the assignment satisfied the clauses. Significantly, we do not seek to define how `CheckSat` checks the satisfiability of the clauses, we only assume that it achieves it in a number of instructions polynomial in n, the number of Boolean variables. We know that SAT is in NP, so due to complexity theory we also know an assignment can be verified in a polynomial number of instructions.[1] We then produce an output, `bit_pattern`, using an "if-then-else" instruction that evaluates to zero if the Boolean variables do not satisfy the clauses, and has all bits set if they do.

For the switching phase, the CSWP instruction counter, `insn_count`, is read to learn how many instructions there are in the decision phase of the CSWP program. We then emit a pattern that repeatedly loads the variable `bit_pattern` and then zero. The effect of this is to produce a phase in the program that causes a large amount of switching if the SAT problem p was satisfied; and to not if it was unsatisfiable. In this sequence, a satisfying assignment will cause the switching phase to flip every bit in the output datapath, every instruction; while no switching will occur otherwise.

We have thus introduced a gap in the switching activity of the CSWP program q, that is governed by whether the SAT problem p is satisfiable or not. We use the length of the decision phase of the program to ensure that the switching phase is at least the length of the decision phase, plus one or two instructions. This ensures that, regardless of the amount of switching in the decision phase, the switching phase dominates the switching activity of the program. When solving CSWP, if the SAT problem p were satisfiable, then the maximum amount of switching would include the switching phase, and the CSWP solver would be obliged to yield an input to the program that satisfied the reduced SAT problem. If p is unsatisfiable, it would instead yield whatever input maximised the switching in the decision phase.

We use the size of the gap to demonstrate that CSWP cannot be approximated. In the previous example the switching phase constitutes at least $1/2$ of the possible switching activity: if one possessed an algorithm to approximate such a CSWP program to a factor of $1/2$, then it would

---

[1]We note that, as the inputs to `CheckSat` are free variables, we are essentially modelling a SAT solver.

be obliged to activate the switching phase of any CSWP program reduced from a satisfiable SAT formula, thus acting as an oracle for an NP-complete problem. Under the P $\neq$ NP assumption, such an algorithm does not exist. $\square$

Furthermore, we are able to extend this result to any constant factor. For any value of ε and SAT instance p, take the desired approximation factor $f = 1 - \varepsilon$ and set the length of the switching phase to be $declen \times (1/f)$, where $declen$ is the number of instructions in the decision phase of CSWP q. Such a program will have a gap of at least $1/f$ times the decision phase, that depends entirely on the satisfiability of p, and thus cannot be approximated. One need not limit this approach to a constant factor either: one may instead compute f to be some factor that is a polynomial function of the size of SAT problem p, for example n^2, and achieve the same result. This shows that there can be no useful approximation factor for CSWP.

The safety of this result depends on the reduction to q being polynomial in the number of variables n in p. Introducing the variables of p scales linearly with n, checking the satisfiability of a particular assignment is known to be checkable in polynomial time, and the evaluation of the result into `bit_pattern` is constant-time. The decision phase is thus a polynomial reduction. The switching phase is controlled by the length of the decision phase (which is polynomial), but also the desired approximation factor. Provided the approximation factor is polynomial, the full reduction is also polynomial.

7 Discussion

We consider here the scope of these results for the analysis of dynamic energy in general, their implications with regard to the feasibility of such analysis, and potential alternative methods for analysing energy in systems.

7.1 Scope

The immediate outcome of these proofs is that, with a program's switching activity shown to be NP-hard, calculating the worst-case dynamic energy for a program is infeasible. Our result relies on the analysis that the cost of switching activity is dominated by switching in the output datapath, meaning the majority of dynamic energy cannot be calculated. Clearly, the exact cost of such switching will vary between processors, however our result may be used as a basis for demonstrating that calculating the switching in other components of the processor is also infeasible. For example, because all inputs to instructions are inevitably the output of some other instruction, it is reasonable to assume that it is NP-hard to estimate the switching activity of input operands too.

Branch prediction and data caches will contribute dynamic energy too. These also depend on program inputs to an extent, but are not modelled by our CSWP formalisation. Other processor components may contribute dynamic energy that is not affected by the inputs to a program — the switching associated with instruction logic (decode, functional unit activation, instruction cache) will contribute dynamic energy regardless of the program input.

Finally, features in some processors, such as out-of-order execution may defeat this analysis. The circuit switching cost is still present, and its determination will still be NP-hard, however it may occur in an unpredictable fashion that depends on a processor-internal unobservable instruction execution schedule.

7.2 Implications

The infeasibility result for estimating dynamic energy over time prevents the construction of an instruction level energy model that identifies a worst case switching cost for each instruction in a given program. Existing techniques that apply WCEC analysis [12, 17, 7, 18, 6, 32] to software can thus never have an energy model that accurately accounts for worst case achievable dynamic energy of the given computation. One may instead, given an accurate model of the switching costs within a processor, assume that every circuit switches in every clock cycle, which will achieve a safe upper bound on the energy consumption [12, 32]. The over-approximation inherent with this approach will not yield a tight bound. For example, on the XMOS XS1-L, with dynamic energy contributing 42 % of energy consumption, one would have a similarly sized amount of potential over-approximation regarding the energy consumption of any execution.

7.3 Alternatives

Viable techniques for estimating dynamic energy consumption can come from a variety of fields: in particular, statistical methods [22] may be effective for determining the distribution of energy consumption under normal operation. Such a model may be used by assuming that the most energy the program can consume occurs only 1 % of the time, and taking the energy value corresponding to that probability as the program's energy consumption. This does not provide a safe upper bound on the program's energy consumption as it is based on normal operation. However, on the balance of probability it is very likely to present an upper bound. Depending on the use case such bound may be more useful in making energy consumption of software transparent to developers than gross over-approximation.

Another alternative is to initially assume that the maximum dynamic energy is dissipated by every instruction, and then use information gained by static analysis to lower this figure. For example, if one can determine the integer interval of a variable, then one can potentially determine the maximum switching of a specific instruction to be lower than its general maximum. The closeness of the bound determined would depend heavily on the capabilities of the static analyses applied, and the extent to which the program lends itself to static analysis. However, our results show that this technique cannot generally provide an accurate bound.

In all circumstances, alternative estimation techniques will posses some level of unsafeness or incompleteness, otherwise they will be NP-hard in the general case as proven in this work.

8 Conclusions

In this paper we have considered the energy consumption in a processor that can directly be attributed to the data or inputs to the software being executed, and demonstrate that the general analysis of circuit switching in processor datapaths — the "circuit switching problem" — is NP-hard. Further, we demonstrate that there is no efficient algorithm for approximating the circuit switching problem to any constant or polynomial factor. We conclude that the analysis of worst-case energy as caused by software cannot be achieved in an efficient manner, leaving a necessary uncertainty factor corresponding to the amount of dynamic energy controlled by processor datapaths. In addition, we consider alternate questions that one could pose that do not amount to worst-case analysis and how they can contribute to understanding software energy consumption.

In the future we believe that work is best focused on statistical methods of modelling program energy consumption, or otherwise characterising the way in which software operates. Critically, we cannot continue to think in terms of "worst case" behaviour, but must instead turn to empirical methods for analysing program behaviours rather than formally proving them.

Acknowledgements

References

[1] G. Ascia et al. "An instruction-level power analysis model with data dependency". English. In: *VLSI DESIGN* 12.2 (2001), 245–273. ISSN: 1065-514X. DOI: {10.1155/2001/82129}.

[2] A. Biere et al. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009, pp. 611–632. ISBN: 1586039296, 9781586039295.

[3] A. Carroll and G. Heiser. "An analysis of power consumption in a smartphone". In: *Proceedings of the 2010 USENIX conference on USENIX annual technical conference.* USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, p. 21. URL: http://portal.acm.org/citation.cfm?id=1855840.1855861.

[4] L. Chandra and S. Roy. "Estimation of energy consumed by software in processor caches". In: *2008 IEEE International Symposium on VLSI Design, Automation and Test (VLSI-DAT).* IEEE, Apr. 2008, pp. 21–24. ISBN: 978-1-4244-1616-5. DOI: 10.1109/VDAT.2008.4542403. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4542403.

[5] A. T. Freitas, H. C. Neto, and A. L. Oliveira. "On the cmoplexity of Power Estimation Problems". 2004.

[6] K. Georgiou, S. Kerrison, and K. Eder. *On the Value and Limits of Multi-level Energy Consumption Static Analysis for Deeply Embedded Single and Multi-threaded Programs.* Tech. rep. University of Bristol, 2015.

[7] N. Grech et al. "Static Analysis of Energy Consumption for LLVM IR Programs". In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems.* SCOPES '15. Sankt Goar, Germany: ACM, 2015, pp. 12–21. ISBN: 978-1-4503-3593-5.

[8] H. Hajimiri, K. Rahmani, and P. Mishra. "Efficient Peak Power Estimation Using Probabilistic Cost-Benefit Analysis". In: *VLSI Design (VLSID), 2015 28th International Conference on.* 2015, pp. 369–374. DOI: 10.1109/VLSID.2015.68.

[9] R. Hameed et al. "Understanding sources of inefficiency in general-purpose chips". In: *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10* (2010), p. 37. DOI: 10.1145/1815961.1815968. URL: http://portal.acm.org/citation.cfm?doid=1815961.1815968.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.

[11] Hsiao et al. "K2: an estimator for peak sustainable power of VLSI circuits". In: *Low Power Electronics and Design* (1997).

[12] R. Jayaseelan, T. Mitra, and X. Li. "Estimating the Worst-Case Energy Consumption of Embedded Software". In: *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium.* RTAS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 81–90. ISBN: 0-7695-2516-4. DOI: 10.1109/RTAS.2006.17. URL: http://dx.doi.org/10.1109/RTAS.2006.17.

[13] D. S. Johnson. "Approximation Algorithms for Combinatorial Problems". In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing.* STOC '73. Austin, Texas, USA: ACM, 1973, pp. 38–49. DOI: 10.1145/800125.804034. URL: http://doi.acm.org/10.1145/800125.804034.

[14] D. Kearney and N. W. Bergmann. "Performance evaluation of asynchronous logic pipelines with data dependent processing delays". In: *Asynchronous Design Methodologies, 1995. Proceedings., Second Working Conference on.* 1995, pp. 4–13. DOI: 10.1109/WCADM.1995.514637.

[15] S. Kerrison and K. Eder. "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor". In: *ACM Trans. Embedded Comput. Syst.* 14.3 (2015), p. 56. DOI: 10.1145/2700104. URL: http://doi.acm.org/10.1145/2700104.

[16] Y. Lee and S. Kim. "DRAM energy reduction by prefetching-based memory traffic clustering". In: *Proceedings of the 21st edition of the great lakes symposium on Great lakes symposium on VLSI - GLSVLSI '11* (2011), p. 103. DOI: 10.1145/1973009.1973031. URL: http://portal.acm.org/citation.cfm?doid=1973009.1973031.

[17] U. Liqat et al. "Energy Consumption Analysis of Programs based on XMOS ISA-Level Models". In: *Logic-Based Program Synthesis and Transformation, 23rd International Symposium, LOPSTR 2013, Revised Selected Papers.* Vol. 8901. Lecture Notes in Computer Science. Springer, 2014, pp. 72–90. ISBN: 978-3-319-14124-4.

[18] U. Liqat et al. "Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR". In: *Proc. of FOPARA.* LNCS. To Appear. Springer, 2015.

[19] D. May. *The XMOS XS1 Architecture. Available online: http://www.xmos.com/published/xmos-xs1-architecture.* 2013.

[20] P. M. Morgado, P. F. Flores, and L. M. Silveira. "Generating Realistic Stimuli for Accurate Power Grid Analysis". In: *ACM Trans. Des. Autom. Electron. Syst.* 14.3 (June 2009), 40:1–40:26. ISSN: 1084-4309. DOI: 10.1145/1529255.1529262. URL: http://doi.acm.org/10.1145/1529255.1529262.

[21] J. Pallister et al. "A high-level model of embedded flash energy consumption". In: *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems - CASES '14.* New York, New York, USA: ACM Press, 2014, pp. 1–9. ISBN: 9781450330503. DOI: 10.1145/2656106.2656108. URL: http://dl.acm.org/citation.cfm?doid=2656106.2656108.

[22] J. Pallister et al. "Data dependent energy modelling: A worst case perspective". In: *CoRR* abs/1505.03374 (2015). URL: http://arxiv.org/abs/1505.03374.

[23] A. Parikh et al. "Instruction Scheduling for Low Power". In: *Journal of VLSI signal processing systems for signal, image and video technology* 37.1 (2004), pp. 129–149. DOI: 10.1023/B:VLSI.0000017007.28247.f6.

[24] S. Rivoire, P. Ranganathan, and C. Kozyrakis. "A Comparison of High-level Full-system Power Models". In: *Proceedings of the 2008 Conference on Power Aware Computing and Systems.* Hot-Power'08. San Diego, California: USENIX Association, 2008, pp. 3–3. URL: http://dl.acm.org/citation.cfm?id=1855610.1855613.

[25] Y. S. Shao and D. Brooks. "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor". In: *International Symposium on Low Power Electronics and Design (ISLPED).* November. IEEE, Sept. 2013, pp. 389–394. ISBN: 978-1-4799-1235-3. DOI: 10.1109/ISLPED.2013.6629328. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6629328.

[26] A. Sinha and A. P. Chandrakasan. "Energy Aware Software". In: *Proceedings of the 13th International Conference on VLSI Design.* VLSID '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 50–. ISBN: 0-7695-0487-6. URL: http://dl.acm.org/citation.cfm?id=580736.835252.

[27] S. Steinke et al. "An Accurate and Fine Grain Instruction-level Energy Model Supporting Software Optimizations". In: *Proceedings of PATMOS.* 2001.

[28] L. Thiele and R. Wilhelm. "Design for Timing Predictability". In: *Real-Time Syst.* 28.2-3 (Nov. 2004), pp. 157–177. ISSN: 0922-6443. DOI: 10.1023/B:TIME.0000045316.66276.6e. URL: http://dx.doi.org/10.1023/B:TIME.0000045316.66276.6e.

[29] V. Tiwari, S. Malik, and A. Wolfe. "Power analysis of embedded software: a first step towards software power minimization". In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 2.4 (1994), pp. 437–445. ISSN: 1063-8210. DOI: `10.1109/92.335012`.

[30] V. Tiwari et al. "Instruction Level Power Analysis and Optimization of Software". In: *J. VLSI Signal Process. Syst.* 13.2-3 (Aug. 1996), pp. 223–238. ISSN: 0922-5773. DOI: `10.1007/BF01130407`. URL: `http://dx.doi.org/10.1007/BF01130407`.

[31] V. V. Vazirani. *Approximation Algorithms.* New York, NY, USA: Springer-Verlag New York, Inc., 2001, pp. 306–311. ISBN: 3-540-65367-8.

[32] P. Wägemann et al. "Worst-Case Energy Consumption Analysis for Energy-Constrained Embedded Systems". In: *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on.* 2015, pp. 105–114. DOI: `10.1109/ECRTS.2015.17`.

[33] R. Wilhelm et al. "The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: `10.1145/1347375.1347389`.