Energy Transparency for Deeply Embedded Programs

KYRIAKOS GEORGIOU and STEVE KERRISON, University of Bristol
ZBIGNIEW CHAMSKI, Infrasoft IT Solutions
KERSTIN EDER, University of Bristol

Energy transparency is a concept that makes a program's energy consumption visible, from hardware up to software, through the different system layers. Such transparency can enable energy optimizations at each layer and between layers, as well as help both programmers and operating systems make energy-aware decisions. In this article, we focus on deeply embedded devices, typically used for Internet of Things (IoT) applications, and demonstrate how to enable energy transparency through existing static resource analysis (SRA) techniques and a new target-agnostic profiling technique, without hardware energy measurements. Our novel mapping technique enables software energy consumption estimations at a higher level than the Instruction Set Architecture (ISA), namely the LLVM intermediate representation (IR) level, and therefore introduces energy transparency directly to the LLVM optimizer. We apply our energy estimation techniques to a comprehensive set of benchmarks, including single- and multithreaded embedded programs from two commonly used concurrency patterns: task farms and pipelines. Using SRA, our LLVM IR results demonstrate a high accuracy with a deviation in the range of 1% from the ISA SRA. Our profiling technique captures the actual energy consumption at the LLVM IR level with an average error of 3%.

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Metrics

General Terms: Algorithms, Measurement, Performance, Reliability

Additional Key Words and Phrases: Static analysis, profiling, deeply embedded systems, IoT, LLVM, WCET

ACM Reference Format:
Kyriakos Georgiou, Steve Kerrison, Zbigniew Chamski, and Kerstin Eder. 2017. Energy transparency for deeply embedded programs. ACM Trans. Archit. Code Optim. 14, 1, Article 8 (March 2017), 26 pages.
DOI: http://dx.doi.org/10.1145/3046679

1. INTRODUCTION

The various abstraction layers introduced through the system stack to make programming easier make it very difficult to understand how coding and data structures affect energy consumption when the program is executed. Energy transparency aims to leverage this information from the lower levels of the system stack up to the user [Eder et al. 2016]. Such information can be of significant value for Internet of Things (IoT) applications, which typically have to operate on limited or unreliable sources of energy.

Deploying millions of embedded devices into IoT environments poses the challenge of how to power them. Battery-based solutions can be costly and impractical due to

The research leading to this work received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement 318337, ENTRA: Whole-Systems Energy Transparency, the European Commission under Grant Agreement No. 611004, ICT-Energy, and the ARTEMIS Joint Undertaking under grant agreement 621429, EMC2.
Authors' addresses: K. Georgiou, S. Kerrison, and K. Eder, Merchant Venturers Building, Woodland Road, Clifton, Bristol, BS8 1UB; emails: {Kyriakos.Georgiou, Steve.Kerrison, Kerstin.Eder}@bristol.ac.uk; Z. Chamski, Infrasoft IT Solutions, Oskara Kolberga 33, 09-407 PLOCK, Poland; email: zbigniew.chamski@gmail.com.

DOI: http://dx.doi.org/10.1145/3046679

the need for replacement. A better solution is a combination of energy harvesting with ultralow energy embedded devices. Energy harvesting comes with two caveats. First, it is an unreliable source of energy; second, it cannot yet deliver the required energy budgets for many IoT applications. Although many achievements have been made in optimizing the energy consumption of hardware, too little is done to expose potential hardware-level energy savings to the software developer. We propose techniques for exposing the bounds and the actual energy consumption of software written for a specific platform, as part of the embedded systems software development cycle. This enables programmers, tool chains, and runtime systems to make energy-aware decisions to meet their strict energy constraints.

The energy consumption of a program on specific hardware can always be determined through physical measurements. Although this is potentially the most accurate method, it is often not easily accessible. Measuring energy consumption can involve sophisticated equipment and special hardware knowledge. Custom modifications may be needed to probe the power supply. Even though energy-monitoring counters are becoming increasingly popular in modern processors, their number and availability are still limited in deeply embedded systems. Moreover, fine-grain energy characterization of software components, such as control flow graph (CFG) basic blocks (BBs) or loops, cannot be achieved by only using energy measurements. All of this makes it difficult for the majority of software developers to assess a program's energy consumption.

Energy consumption is a resource constraint; another frequently examined constraint is execution time. Significant progress has been made in the area of worst-case execution time (WCET) prediction using static resource analysis (SRA) techniques that determine safe upper bounds for the execution time of programs. A popular approach used for WCET is the implicit path enumeration technique (IPET), which retrieves the worst-case control flow path of programs based on a timing cost model. Instead, in Jayaseelan et al. [2006], an energy model that assigns energy values to blocks of instruction set architecture (ISA) code is used to statically estimate worst-case energy consumption (WCEC). We also adopted IPET in our SRA, retrieving energy bounds for the processor under investigation, the XMOS XS1-L "Xcore."

The Xcore is a multithreaded deeply embedded processor with time-deterministic instruction execution. Such systems are simpler than general-purpose processors and favor predictability and low energy consumption over maximizing performance. Processors with such characteristics are the backbone for IoT applications. Moreover, the absence of performance-enhancing complexity at the hardware level, such as caches, make them ideal for critical applications. We base our choice of the Xcore processor among other more popular architectures used for IoT applications, such as the `ARM Cortex-M` series [ARM 2016], on the fact that the Xcore is a time-deterministic multithreaded architecture that can be extended to many-core systems [Hollis and Kerrison 2016], allowing for a variety of design space exploration choices. Our SRA uses an ISA multithreaded energy model for the Xcore introduced in Kerrison and Eder [2015].

In addition, we have developed a novel mapping technique to lift our ISA-level energy model to a higher level, the intermediate representation (IR) of the compiler, namely LLVM IR [Lattner and Adve 2004], implemented within the LLVM tool chain [LLVMorg 2014]. This enables SRA to be performed at a higher abstraction level than ISA, thus introducing energy transparency into the compiler tool chain by making energy consumption information accessible directly to the optimizer. Transparency of energy consumption at this level enables programmers to investigate how optimizations affect their program's energy consumption [Blackmore et al. 2015], or even helps to introduce new low energy optimizations [Pallister et al. 2014, 2015a]. This is more applicable at the LLVM IR than at the ISA level, because more program information exists at that level, such as types and loop structures. Our mapping and analysis techniques at the LLVM IR level are applicable to any compiler that uses the LLVM common optimizer, provided that an energy model for the target architecture is available. Our LLVM IR analysis results demonstrate a high accuracy with a deviation in the range of 1% from the ISA SRA.

Considering that there is no practical method to perform actual case static analysis at this time [Townley 2013], we introduce a profiling technique that can provide actual energy estimations directly at the LLVM IR level. The profiler is implemented at the LLVM IR level to keep the technique as target-agnostic as possible. Moreover, the technique is more favorable than instruction set simulation (ISS)-based estimations, as building an ISS for an architecture is a significantly bigger task. Our profiling-based estimation can provide at least the same performance or significantly outperform the estimation speed of an ISS-based estimation, depending on the complexity of the algorithm implemented and the size of the resulting program. This makes it more suitable for iterative optimizations during software development. The profiling-based estimation is evaluated on a large set of benchmarks, showing an only 1.8% deviation compared to a cycle-accurate ISS for the Xcore.

The main contributions of this article are the following:

(1) Formalization and implementation of a novel target-agnostic mapping technique that lifts an ISA-level energy model to a higher level, the IR of the LLVM compiler (Section 3).
(2) SRA energy estimation at the ISA level and at the LLVM IR level using our mapping technique (Section 4.1).
(3) A new target-agnostic profiling-based energy estimation technique that retrieves estimations at the LLVM IR level by the use of our mapping technique, and with an accuracy close to a cycle-accurate ISS-based estimation (Section 4.2).
(4) SRA and profiling extension for a set of multithreaded programs (Section 4.3), focusing on task farms and pipelines, which are two commonly used concurrency patterns.
(5) Comprehensive evaluation of our SRA and profiling energy estimation techniques and our mapping technique accuracy on a large set of benchmarks (Section 5).

The rest of the article is organized as follows. Section 2 gives an overview of the Xcore architecture and its ISA energy model. Section 3 introduces our mapping technique and its instantiation for the Xcore processor. Section 4 details our two estimation methods: SRA and profiling based. Our experimental evaluation methodology, benchmarks, and results are presented and discussed in Section 5. Section 6 critically reviews previous work related to ours. Finally, Section 7 concludes the article and outlines opportunities for future work.

2. XCORE ARCHITECTURE AND ENERGY MODEL

2.1. Xcore Architecture Overview

The Xcore processor is a deeply embedded processor intended to allow hardware interfaces to be implemented in software. This makes the Xcore well suited to embedded applications requiring multiple hardware interfaces with real-time responsiveness. Interfaces, such as SPI, I2C, and USB, can be written efficiently in C-style software rather than relying on hardware blocks provided in a system-on-chip or synthesized onto FPGA. To achieve this, the Xcore is designed to be a time-deterministic hardware multithreaded architecture that provides interthread communication and I/O port control directly in the ISA. Moreover, for energy efficiency, the processor is event driven; busy waiting is avoided in favor of hardware-scheduled idle periods. The processor supports up to eight threads, with machine instructions and hardware resources dedicated to thread creation, synchronization, and destruction. Each thread has its own instruction buffer and register bank. The pipeline of the processor was designed to provide time-deterministic execution and to maximize responsiveness, making the processor ideal for IoT applications. It is integer only, with no floating point hardware. By design, the architecture avoids the need for forwarding between pipeline stages, speculative instruction issue, and branch prediction and allows for zero time overhead in thread context switching. Threads are executed round-robin through a four-stage pipeline. Each thread can have only one instruction occupy the pipeline at any time, avoiding data hazards. Therefore, if fewer than four threads are active, there will be clock cycles with inactive pipeline stages. This means that to reach the maximum computation power of the processor, four or more threads have to be active to fully occupy the processor's pipeline. When more than four threads are active, maximum throughput is maintained, but compute time is divided between active threads.

The Xcore is many-core scalable, with two- or five-wire "X-Links" and a network switch embedded into each core. Communication between threads on the same or different cores is done via synchronous channel-based message passing. Threads on the same core can communicate without contention, whereas the links used for multicore communication may be contended. These properties allow design space exploration of multithreaded programs that are core-local and/or multicore. A case of the trade-off between processor count and energy-efficient execution is investigated in Section 5.1.2. Full details of the architecture are provided in May [2009].

2.2. Xcore Multithreaded ISA Energy Model

The underlying energy model for this work is captured at the ISA level. Individual instructions from the ISA are assigned a single cost each. These can then be used to compute power or energy for sequences of instructions. The modeling technique is built upon Tiwari et al. [1996], which is adapted and extended to consider the scheduling behavior and pipeline characteristics of the Xcore [Kerrison and Eder 2015]. The model captures the cost of thread scheduling performed by the hardware in accordance with a series of profiling tests and measurements, because it influences the energy consumption of program execution. The cost associated with an instruction represents the average energy consumption obtained from measuring the energy consumed during instruction execution based on constrained pseudorandomly generated operands. A new version of this model that is well suited for static analysis has been developed. It represents energy in terms of static and dynamic power components to better reflect interinstruction and interthread overheads. This has improved model accuracy by an average of four percentage points.

$$E_{prg} = (P_s + P_{di}) \cdot T_{idl} + \sum_{i \in prg} \left(\frac{P_s + P_i M_{N_p} O}{N_p} \cdot 4 \cdot T_{clk} \right), \quad \text{where } N_p = \min(N_t, 4) \quad (1)$$

In Equation (1), E_{prg} is the energy of a program formed by adding the energy consumed at idle to the energy consumed by every instruction, i, executed in the program. At idle, only a base processor power, the sum of its static power, P_s, and dynamic idle power, P_{di}, is dissipated for the total idle time, T_{idl}. For each instruction, static power is again considered, with additional dynamic power for each particular instruction, P_i. The dynamic power contribution is then multiplied by a constant interinstruction overhead, O, which has been established as the average overhead of instruction interleaving. This is then multiplied by a scaling factor to account for the number of threads in the pipeline, M_{N_p}. The result is divided by the number of instructions in the pipeline, which is at most four and is dependent upon the number of active threads, N_t. Each instruction completes in four cycles, so $4 \cdot T_{clk}$ gives the energy contribution of the given instruction based on the calculated power.

When more than four threads are active, the issue rate of instructions per thread will be reduced. The energy model accounts for this with the min term in Equation (1). From a purely timing perspective, the latency between instruction issues for a thread is $\max(N_t, 4) \cdot T_{clk}$. This property means that instructions are time deterministic, provided the number of active threads is known. A thread may stall to fetch the next instruction. This is also deterministic and can be statically identified [May 2009, pp. 8–10]. These instruction timing rules have been used in simulation-based energy estimation and are also utilized in both the SRA and profiling performed in this article.

A limited number of instructions can be exceptions to these timing rules. The divide and remainder instructions are bit serial and take up to 32 cycles to complete. Resource instructions may block if a condition of their execution is not met. For example, waiting on inbound communication causes the instruction's thread to be de-scheduled until the condition becomes satisfied. This work focuses its contributions on fully predictable instructions, with timing disturbances from communication forming future work.

2.3. Utilizing the Xcore Energy Model for Energy Consumption Estimations

To determine the energy consumption of a program based on Equation (1), the program's instruction sequence, $\langle i_1, \ldots, i_n \rangle$, the idle time T_{idl}, and the number of active threads N_p during instruction execution must be known. In Kerrison and Eder [2015], an ISS was used to gather full trace data or execution statistics to obtain these parameters. In this work, we use ISS only as a reference for comparison of SRA and profiling results, with a second reference being direct hardware measurement.

For both SRA and profiling, we need to extract the CFGs for each thread and identify the interleavings between them. This allows for each instruction in the program to identify the N_p component in Equation (1). It also allows estimation of the total idle time, T_{idl}, of the program. For single-threaded programs, the energy characterization of the CFG is straightforward, as there is no thread interleaving. For SRA, the IPET can be directly applied to the energy-characterized CFG to extract a path that bounds the energy consumption of the program, as described in Section 4.1. For arbitrary multithreaded programs, using static analysis to characterize the CFG of each thread with respect to energy consumption is challenging. We have therefore concentrated on two commonly used concurrency patterns, task farms and pipelines, which we use with evenly distributed workloads across threads. For these classes of programs, the number of active threads across the whole execution is constant and equal to the number of threads used to implement the task farm or the pipeline. Similarly, the profiling technique does not need to account for thread interleavings for the programs investigated in this work.

In addition to instructions defined in the ISA, a fetch no-op (FNOP) can also be issued by the processor. These occur deterministically [May 2009, pp. 8–10]. FNOPs can significantly impact on energy consumption, particularly within loops. To account for FNOPs in both our SRA and profiling, the program's CFG at the ISA level is analyzed. An instruction buffer model is used to determine where FNOPs will occur in a BB. Further details on FNOP modeling can be found in Georgiou [2016b].

3. MAPPING ISA CODE TO LLVM IR AND LLVM IR ENERGY CHARACTERIZATION

Our mapping technique aims to link each LLVM IR instruction of a program with its corresponding machine-specific ISA-emitted instructions. Such a mapping can give powerful insights to the LLVM IR optimizer regarding code size, execution time, and the energy consumption of a program. Furthermore, our LLVM mapping technique does not involve statistical analysis; instead, it is an on-the-fly technique that takes into consideration the compiler behavior and the actual program's CFG structure. The technique is fully portable and target agnostic. It requires only the adjustment of the LLVM mapping pass to the new architecture.

In this section, we first formalize our generic mapping technique. We then specialize the technique to determine the energy characteristics of LLVM IR instructions. This specialization propagates ISA-level energy models up to the LLVM IR level, enabling energy consumption estimation of programs at that level. Finally, we instantiate and tune the mapping technique for the architecture under consideration—the Xcore.

3.1. Formal Specification of the Mapping

The main idea of the mapping technique is to monitor the back-end of a compiler to establish a $1 : m$ relation between the optimized LLVM IR and the emitted ISA. The goal of this mapping is to associate a single LLVM IR instruction with all ISA instructions that originated from it, whenever possible. Then, by aggregating the energy costs of these ISA instructions, we can assign an energy cost to their single corresponding LLVM IR instruction. The mapping technique also guarantees that there is no loss of energy between the two levels, as each ISA instruction will be mapped to one LLVM IR instruction. We formalize the mapping as follows. For a program P, let

$$\mathrm{IRprog}_L = \{1, 2, \ldots, n\} \tag{2}$$

be the ID numbers of P's LLVM IR instructions after the LLVM transformation and optimizations passes, with

$$\mathrm{IRprog} = \langle ir_1, ir_2, \ldots, ir_n \rangle \tag{3}$$

being the sequence of LLVM IR instructions for P. An architecture-specific compiler back-end T_{arch} translates the IR program into ISA code:

$$\mathrm{T}_{arch}(\mathrm{IRprog}) = \mathrm{ISAprog} = \langle isa_1, isa_2, \ldots, isa_k \rangle, \tag{4}$$

producing a sequence of machine instructions $\langle isa_1, isa_2, \ldots, isa_k \rangle$ that represents the program P at the ISA level. Let

$$\mathrm{M}(\mathrm{IRprog}) = \langle (isa_1, m_1), (isa_2, m_2), \ldots, (isa_k, m_l) \rangle \text{ where } m_1, m_2, \ldots, m_l \in \mathrm{IRprog}_L \tag{5}$$

be the mapping process that monitors T_{arch} and creates a relation between the sequence of ISA instructions for P and the IDs of the LLVM IR instructions, with the aim to associate an ISA instruction with the LLVM IR instruction from which it originated, whenever this is possible. For ISA instructions that are not related to any LLVM IR instruction (ISA-injected instructions) or whose origin LLVM IR instruction is ambiguous, the mapping process has to make an implementation-specific choice as to which LLVM IR instruction an ISA instruction should be associated with. This will preserve the $1 : m$ relation and ensure that such instructions are accounted at the LLVM IR level. The mapping function

$$\begin{gathered}\mathrm{R}(ir_i) = \{isa_j | ir_i \in \mathrm{IRprog} \wedge isa_j \in \mathrm{ISAprog} \wedge (isa_j, i) \in \mathrm{M}(\mathrm{IRprog})\} \text{ with} \\ \text{the property } \forall\, ir_n, ir_k \in \mathrm{IRprog} \wedge n \neq k \text{ then } \mathrm{R}(ir_n) \cap \mathrm{R}(ir_k) = \emptyset\end{gathered} \tag{6}$$

captures a $1 : m$ relation from IRprog to ISAprog instructions. The energy consumption of an LLVM IR instruction can be retrieved by

$$\mathrm{E}(ir_i) = \sum_{isa_j \in S} \mathrm{E}(isa_j) \text{ where } ir_i \in \mathrm{IRprog} \wedge isa_j \in \mathrm{ISAprog} \wedge \mathrm{S} = \mathrm{R}(ir_i) \tag{7}$$

as the sum of the energy consumed by all ISA instructions mapped to that LLVM IR instruction. Equation (7) can also be used to associate timing or code size information with LLVM IR instructions by replacing the ISA instructions' energy costings with the resource of interest costings.

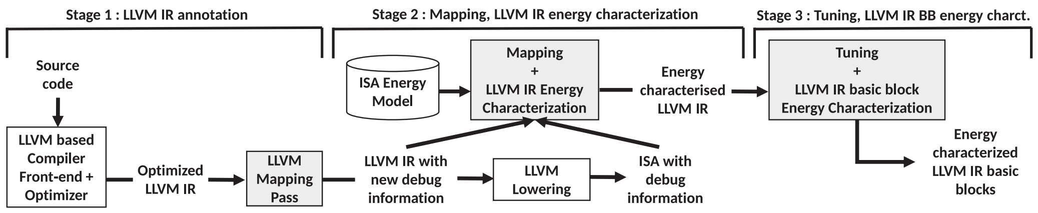

Stage 1 : LLVM IR annotation
Stage 2 : Mapping, LLVM IR energy characterization
Stage 3 : Tuning, LLVM IR BB energy charct.
Source code
LLVM based Compiler Front-end + Optimizer
Optimized LLVM IR
LLVM Mapping Pass
ISA Energy Model
Mapping + LLVM IR Energy Characterization
Energy characterised LLVM IR
LLVM IR with new debug information
LLVM Lowering
ISA with debug information
Tuning + LLVM IR basic block Energy Characterization
Energy characterized LLVM IR basic blocks

Fig. 1. LLVM IR energy characterization overview.

3.2. Xcore Mapping Instantiation and Tuning

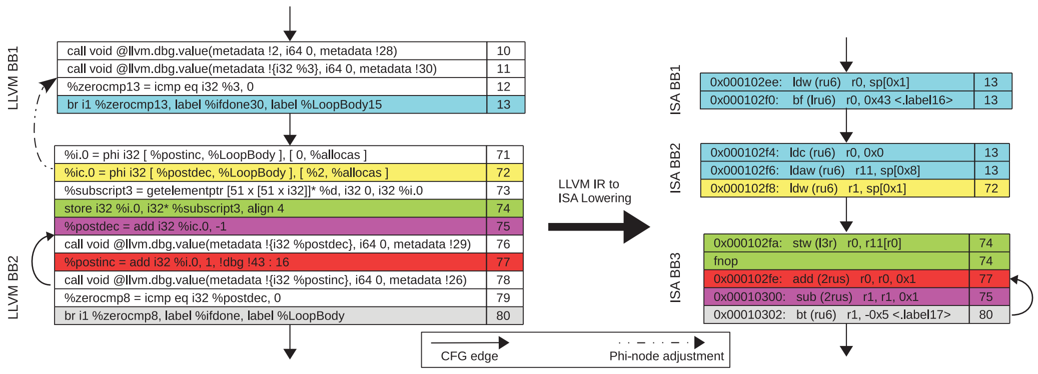
In our case, the T_{arch} function is the XMOS tool chain lowering phase that translates the LLVM IR to Xcore-specific ISA. The real challenge is to create from scratch a mechanism that can monitor T_{arch} and create the mapping given by Equation (5) that will have the property of Equation (6). This would require an extensive knowledge of the back-end of the architecture under consideration and a vast engineering effort to take into account every possible transformation and optimization that happens between the optimized LLVM IR and the final ISA-emitted code. Instead, we demonstrate a simple yet powerful technique that can provide sufficiently accurate results. An overview of the technique is given in Figure 1. We now describe the three mapping stages.

3.2.1. Stage 1: LLVM IR Annotation. The goal of this stage is to enable the property of Equation (6) that ensures a $1 : m$ relation between the LLVM IR and ISA code. To achieve this, our mapping implementation leverages the debug mechanism in the XMOS compiler tool chain. Symbols are created during compilation to assist with debugging. These symbols are propagated to all intermediate code layers and down to the ISA code. Debug symbols can express which programming language constructs generated a specific piece of machine code in a given executable module. In our case, these symbols are generated by the front-end of the XMOS compiler in standard DWARF format [DWARF 2013]. These are transformed to LLVM metadata [Lattner and Patel 2014] and attached to the LLVM IR. LLVM 2.7 and upward use this metadata format as the primary means of storing debug information.

During the lowering phase of compilation, LLVM IR code is transformed to a target ISA by the back-end of the compiler, with debug information stored alongside in the DWARF standard format. Tracking source code debug information gives an $n : m$ relationship between instructions at the different layers, because several source code instructions can be translated to many LLVM IR instructions, and these again into many ISA instructions. This $n : m$ relation prevents the fine-grain energy mapping needed for accurate energy estimations.

To achieve a $1 : m$ mapping between the optimized LLVM IR instructions and the ISA instructions using the debug mechanism, we created an LLVM pass that traverses the optimized LLVM IR and replaces source location information with LLVM IR location information, or adds new location information to LLVM IR instructions without any debug data. This LLVM pass runs after all optimization passes, just before emitting ISA code, because the optimized LLVM IR is closer in structure to the ISA code than the unoptimized version. An example output of this process is given on the left-hand side of Figure 2. This represents a part of the LLVM IR CFG of a program after the LLVM optimizations, along with the unique debug location, IRprog_L in Equation (2), assigned to each LLVM IR instruction.

The XMOS compiler debug mechanism applies the following four rules to preserve the debug information through the different transformations and optimizations applied in the back-end:



```
call void @llvm.dbg.value(metadata !2, i64 0, metadata !28)                  10
call void @llvm.dbg.value(metadata !{i32 %3}, i64 0, metadata !30)           11
%zerocmp13 = icmp eq i32 %3, 0                                               12
br i1 %zerocmp13, label %ifdone30, label %LoopBody15                         13

%i.0 = phi i32 [ %postinc, %LoopBody ], [ 0, %allocas ]                      71
%ic.0 = phi i32 [ %postdec, %LoopBody ], [ %2, %allocas ]                    72
%subscript3 = getelementptr [51 x [51 x i32]]* %d, i32 0, i32 %i.0           73
store i32 %i.0, i32* %subscript3, align 4                                    74
%postdec = add i32 %ic.0, -1                                                 75
call void @llvm.dbg.value(metadata !{i32 %postdec}, i64 0, metadata !29)     76
%postinc = add i32 %i.0, 1, !dbg !43 : 16                                    77
call void @llvm.dbg.value(metadata !{i32 %postinc}, i64 0, metadata !26)     78
%zerocmp8 = icmp eq i32 %postdec, 0                                          79
br i1 %zerocmp8, label %ifdone, label %LoopBody                              80
```

```
0x000102ee:   ldw (ru6)   r0, sp[0x1]                13
0x000102f0:   bf (lru6)   r0, 0x43 <.label16>        13

0x000102f4:   ldc (ru6)   r0, 0x0                    13
0x000102f6:   ldaw (ru6)   r11, sp[0x8]              13
0x000102f8:   ldw (ru6)   r1, sp[0x1]                72

0x000102fa:   stw (l3r)   r0, r11[r0]                74
fnop                                                 74
0x000102fe:   add (2rus)   r0, r0, 0x1               77
0x00010300:   sub (2rus)   r1, r1, 0x1               75
0x00010302:   bt (ru6)   r1, -0x5 <.label17>         80
```

Fig. 2. Fine-grain $1 : m$ mapping including our LLVM mapping pass.

(1) If an instruction is eliminated, its debug location information is also eliminated.
(2) If one instruction is transformed into another one, the debug location of the original instruction is assigned to the new instruction.
(3) If multiple instructions are merged into one, the debug location of one of the initial instructions, the first one in our case, is assigned to the new instruction.
(4) If one instruction is transformed into multiple ones, then all new instructions are being assigned the debug location of their origin instruction.

These rules iteratively preserve the $1 : m$ relationship between the LLVM IR instructions in IRprog and the final ISA in ISAprog, with two exceptions.

The first is when instructions are introduced at the ISA level and they do not correspond to any LLVM IR instruction (ISA-injected instructions). In such cases, the mapping process can assign to them the debug location of an adjacent ISA instruction in the same BB. This ensures that they are accounted for in the mapped LLVM IR block. The second is when a transformation takes as an input multiple instructions and converts them to another set of multiple instructions in a single step. An example of such transformations are peephole optimizations. In such cases, the handling of debug information depends on the compiler implementation. For the back-end under investigation, ISA instructions generated by such transformations are left without any debug information. To account for these instructions at the LLVM IR level, we use the same approach as for ISA-injected instructions.

3.2.2. Stage 2: Mapping and LLVM IR Energy Characterization. Once the LLVM IR annotation has been performed for a program, the back-end lowering phase translates the LLVM IR code to target-specific ISA code. Then the mapping phase, which implements Equation (6), runs and maps LLVM IR instructions with the new debug locations to the emitted ISA instructions that carry the same debug location ID. Finally, the energy values for groups of ISA instructions are aggregated and then associated with their single corresponding LLVM IR instruction, as described by Equation (7).

An example mapping is given in Figure 2. On the left-hand side is a part of the LLVM IR CFG of a program, which represents the IRprog in Equation (3), along with the new debug location assigned to each LLVM IR instruction by the mapping pass and represented by Equation (2). The right-hand side shows the corresponding ISA CFG, together with the debug locations for each ISA instruction, given by Equation (5). The coloring of the instructions demonstrates the mapping between the two CFGs' instructions using Equation (6). Now, one LLVM IR instruction is associated with many ISA instructions, but each ISA instruction is mapped to only one LLVM IR instruction. Some LLVM IR instructions are not mapped because they are removed during the lowering phase of the compiler. This mapping also guarantees that all ISA instructions are mapped to the LLVM IR, so there is no loss of energy between the two levels.

This Xcore instantiation of the mapping technique, using the debug mechanism, can be ported to any architecture supported by the LLVM IR compiler for which an ISA-level energy model exists. Typically, these are deeply embedded processors, such as the one used here and the `ARM Cortex M` series, which are ideal for IoT applications.

3.2.3. Stage 3: Tuning and BB Energy Characterization. Without any further tuning, the mapping instantiation for the Xcore architecture provides an average deviation of 6% between the SRA prediction at the LLVM IR level and that at the ISA level. An additional tuning phase is introduced after the mapping to account for specific architecture behavior and to facilitate our BB-level energy analysis. In the case of the bound analysis, this improved the LLVM IR SRA accuracy, narrowing the gap between ISA and LLVM IR energy predictions to an average of 1% as demonstrated in our results in Section 5.1. This tuning had a similar positive effect on the accuracy of the LLVM IR profiling–based energy estimation.

As discussed in Section 2.3, FNOPs can be issued by the processor and statically determined at the ISA level. This cannot be represented in LLVM IR. Ignoring FNOPs can therefore lead to a significant underestimation of energy at the LLVM IR level. To account for FNOPs at the LLVM IR level, we treat them similarly to the ISA-injected instructions—by assigning them the debug location of an adjacent ISA instruction in the same BB. Figure 2 provides an example of such FNOP treatment at debug location number 74.

To estimate energy at the LLVM BB level, the energy cost of each BB is needed. This can be obtained by accumulating the energy costs of all LLVM IR instructions in an LLVM IR BB. When estimating energy at the BB level, the position of the LLVM IR instructions in BBs is critical, and tuning may be needed to transfer mapped energy costs to different BBs to reflect more accurately where the energy is consumed during program execution. *Phi-nodes*, for example, benefit from such tuning.

Phi-nodes can be introduced at the start of a BB as a side effect of the single static assignment (SSA) used for variables in the LLVM IR. A phi node takes a list of pairs, where each pair contains a reference to the predecessor block together with the variable that is propagated from there to the current block. The number of pairs is equal to the number of predecessor blocks of the current block. A phi node can create inaccuracies in the mapping when LLVM IR is lowered to ISA code that no longer supports SSA because it can be hoisted out from its current block to the corresponding predecessor block at the ISA level. For blocks in loops, this can lead to a significant estimation error. Such cases can be detected by examining the mapping. Similar inaccuracies can be introduced by branching LLVM IR instructions with multiple targets if the ISA of the target processor supports only single-target branches.

Algorithm 1 detects cases where the ISA energy values mapped to phi nodes in a looping BB are accumulated into the wrong LLVM IR BB. It then hoist these costs out to the appropriate LLVM IR predecessor BBs. The algorithm detects the problematic cases by using the mapping and the BB loop depth. Then, by examining the mapping, function `FindIRBB` at line 10 finds the LLVM IR predecessor BB that matches the ISA block holding the ISA phi node–generated instruction. The ISA's instruction energy cost is then added to the cost of the selected predecessor LLVM IR BB and subtracted from the phi node's LLVM IR BB. Performing the mapping after the LLVM IR optimization passes and just before the lowering phase increases the possibility of similar CFG structures between the two levels. This improves the ability of `FindIRBB` to find the correct BB and therefore improves the results of the phi node tuning.

ALGORITHM 1: Phi Node Tuning

```
input: IRprog for a program $P$ as described by Equation (3)
input: IRprogCFG, the CFG for IRprog, with the total energy cost for each BB
input: ISAprogCFG, the CFG for the ISAprog given by Equation (4) with BB total energy info
input: $M$, the $1 : m$ mapping retrieved for $P$ using the mapping as described in this section
for each $ir_n \in$ IRprog do
    if $ir_n$ is a phi-node then
        ISAmap ← GetISAmap($ir_n$, $M$) // all the ISA instructions mapped to $ir_n$
        irBBloopDepth ← GetInstrBBLoopDepth($ir_n$) // degree of nested loops for $ir_n$'s BB
        FromIRBB ← GetCurrentIRBB (IRprogCFG, $ir_n$) // the $ir_n$'s BB
        for each $isa_k \in$ ISAmap do
            isaBBloopDepth ← GetInstrBBLoopDepth($isa_k$) // degree of nested loops for $isa_k$'s BB
            if irBBloopDepth > isaBBloopDepth then
                ISABB ← GetISABB (ISAprogCFG, $isa_k$) // the $isa_k$'s BB
                ToIRBB ← FindIRBB (IRprogCFG, ISABB) // finds the IR predecessor BB of FromIRBB
                that matches ISABB by examining the mapping of their instructions
                if ToIRBB not NULL then
                    ISAcost ← GetCost ($isa_k$) // the energy cost of $isa_k$ from our ISA energy model
                    RemoveCostFromBB (FromIRBB, ISAcost)// remove cost from initial block
                    AddCostToBB (ToIRBB, ISAcost) // add cost to the chosen predecessor block
```

An example of a phi node adjustment is given in Figure 2 at debug location 72. Its corresponding ISA instruction is hoisted out from the loop BB, `ISA BB3`, and into `ISA BB2`. A similar hoisting is performed from `LLVM BB2` and into `LLVM BB1` by Algorithm 1, thus correctly assigning energy values to each LLVM IR block.

3.3. Limitations

Our mapping approach between the LLVM IR and the ISA guarantees that no energy is lost between the two levels, as all ISA instruction energy costs are propagated to the LLVM IR level. Therefore, any inaccuracies introduced in our LLVM IR energy estimations from the mapping are a consequence of attributing ISA energy costs to the wrong LLVM IR BBs. This is because our LLVM IR estimation techniques work at a BB level. In that respect, two cases can affect the estimation accuracy.

In the first case, for instructions at the boundaries of LLVM IR BBs, their corresponding ISA instructions may cross these boundaries at the ISA level. The mapping will, however, still associate the energy costs of these ISA instructions with their original LLVM IR instruction, and thus with their original LLVM IR BB. If such LLVM IR instructions belong to a BB that is part of a loop, when the ISA instruction has been hoisted out of that loop, then, with no proper adjustment, an overestimation will occur due to the mapping. An example of such a case are the phi node instructions, described in Section 3.2.3, where Algorithm 1 is introduced to adjust their mapping. Such cases can be statically identified by examining the mapping. In the second case, a difference between the two CFG structures can occur when BBs are introduced/eliminated at the ISA level. If this makes the structures of the two CFGs significantly different, then the energy costs allocated to the LLVM IR BBs by the mapping can be inaccurate. Performing the mapping after the LLVM IR optimization passes and just before the lowering phase increases the possibility of having similar CFG structures between the two levels and therefore the accuracy of the mapping. The impact of the preceding issues is investigated in Section 5.

3.4. Discussion

The mapping technique uses an ISA resource model. This has significant benefits over a stand-alone LLVM IR static energy model. First, our mapping-based approach benefits from the accuracy that ISA models can provide because the ISA is closer to the hardware than LLVM IR. Second, the dynamic nature of the mapping technique can account for specific architecture behavior, such as the LLVM IR location to which the costs of the FNOPs should be attributed, and compiler-specific behavior, such as code transformations. Static IR energy models that are created through statistical approaches are inherently limited in their ability to account for these.

Furthermore, since Tiwari et al. [1996], the seminal approach of constructing ISA energy models, there are several well-defined ISA energy models [Sarta et al. 1999; Brooks et al. 2000; Steinke et al. 2001; Sami et al. 2002; Ibrahim et al. 2008]. These models could now be lifted to the compilers' IR by our mapping. Therefore, our approach benefits from a well-understood process by which energy models can be created for deeply embedded systems, where predictability is provided at the ISA level.

4. ENERGY ESTIMATION METHODS USED

4.1. Static Resource Analysis

Our IPET-based SRA is implemented in three stages, which is detailed as follows:

(1) *Low-level analysis*: This stage aims to model the dynamic behavior of the processor's microarchitecture. For our energy consumption analysis, this is achieved through the ISA-level energy model detailed in Section 2.2, which captures the behavior of the Xcore processor with regard to its energy consumption characteristics. For the LLVM IR analysis, an extra step is required to characterize the energy consumed by LLVM IR instructions as detailed in Section 3.2.

(2) *Control flow analysis*: This stage aims to capture the dynamic behavior of the program and associates CFG BBs with the information needed for the computation step of the analysis. IPET requires the CFG and call graph of a program to be constructed at the same level as the analysis. At LLVM IR level, the compiler can generate them. At the ISA level, a tool was created to construct them. To detect BBs that belong to a loop or recursion, we adopted and extended the algorithm in Wei et al. [2007]. The CFGs are annotated according to the needs of the IPET described in Li and Malik [1997]. Finally, the annotated CFGs are used in the computation step to produce integer linear programming (ILP) formulations and constraints.

(3) *Computation*: The IPET adopted in our work to estimate the energy consumption of a program is based on Li and Malik [1997]. To infer the energy consumption of a program, instead of using the time cost of a CFG BB, the BB's energy cost is used. The minimum required user input to enable bounding of the problem is the loop bounds declaration. This is also standard practice in timing analysis [Wilhelm et al. 2008]. Further constraints, such as denoting CFG infeasible paths, can be provided in the form of user source code annotations to extract more accurate estimations. The annotation language used in this work can be found in Eder et al. [2013].

4.2. Profiling-Based Energy Consumption Estimation

In contrast to SRA, given a specific set of input parameters, our profiling technique aims to provide the actual energy consumed by a program rather than bounds. To achieve this, the technique collects LLVM IR BB execution counts. The technique is target agnostic in the sense that the instrumentation code is inserted at the architecture-independent LLVM IR level and does not require the modification of a program's object code to insert instrumentation instructions, unlike in other approaches [Srivastava and Eustace 1994]. The energy consumption estimations are also collected at the LLVM IR level. This is enabled by the energy characterization of the LLVM IR code using the mapping technique introduced in Section 3.

Figure 3 outlines the profiling-based energy consumption estimation process. The process is split in two phases: a profiling phase, which aims to collect BB execution

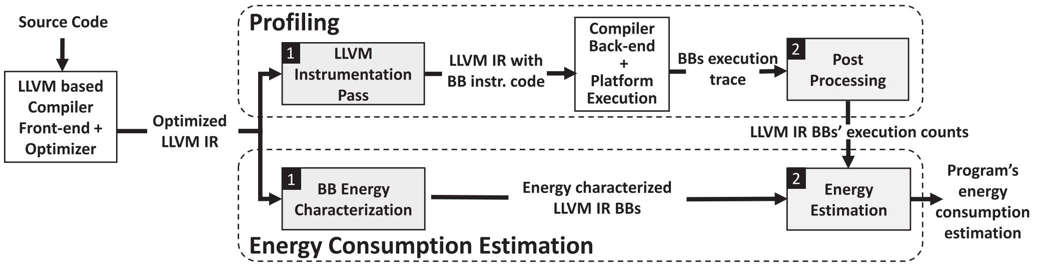

Source Code
LLVM based Compiler Front-end + Optimizer
Optimized LLVM IR
Profiling
1 LLVM Instrumentation Pass
LLVM IR with BB instr. code
Compiler Back-end + Platform Execution
BBs execution trace
2 Post Processing
LLVM IR BBs' execution counts
1 BB Energy Characterization
Energy characterized LLVM IR BBs
2 Energy Estimation
Program's energy consumption estimation
Energy Consumption Estimation

Fig. 3. Profiling-based energy estimation.

traces, and an energy estimation phase, which performs the LLVM IR energy characterization and combines it with the profiling information to obtain energy estimations. Both phases take as input the same LLVM IR, the one retrieved after all optimization passes and just before emitting the target's ISA code. Using the optimized LLVM IR allows us to benefit from a more accurate energy characterization via the mapping technique, as it is closer in structure to the ISA code than the unoptimized version. This also avoids possible negative impact on the LLVM optimizations' abilities, resulting from the injected instrumentation code. The stages for each of the two phases are annotated with numbers in black boxes in Figure 3 and are explained next.

Profiling Phase

(1) *LLVM instrumentation pass*: A new LLVM pass is built into the LLVM compiler that runs after all of the LLVM optimization passes and just before the ISA lowering. The pass injects each LLVM IR BB with instrumentation instructions. To implement the BB instrumentation, we considered two choices. First, using BB execution counters and second using instructions that emit a unique ID, identifying the BB and its origin function, every time the BB is executed. The second option puts less stress on the limited memory size, typically available on deeply embedded devices, as there is no need to keep global variables. Therefore, we chose to use the *print* instruction supported in the LLVM IR assembly language. These *print* instructions need to be translated by the compiler back-end to target-specific tracing functions. In the Xcore, *printf* real-time debugging is supported by the `xTAG` debug adapter [XMOS 2016a], which emits the data to the host machine via buffering, with negligible impact on program execution time. The only requirement for other embedded devices to benefit from our profiler is the implementation of a target-specific instrumentation function to emit profiling data to the host machine, if not already in place. On most targets, this can be implemented using I/O ports, with very low impact on the program's execution time.

(2) *Postprocessing*: The program is then executed and the BBs' execution trace is collected. Based on the unique IDs emitted, the postprocessing stage can associate execution counts with each BB of the optimized LLVM IR code.

Energy Consumption Estimation Phase

(1) *BB energy characterization*: A clean copy of the optimized LLVM IR code, with no instrumentation instructions, is energy characterized using the BB energy characterization mapping techniques introduced in Section 3. This ensures that no energy overheads appear in our energy consumption estimations due to instrumentation code.

(2) *Energy estimation*: This stage takes as input the BB energy-characterized LLVM IR and the BBs' execution counts collected from the profiling phase, and produces the program's total energy consumption estimation. Having the LLVM IR BBs' execution counts and the LLVM IR instruction energy costings, a fine-grain software energy

characterization can be achieved, where we can attribute energy consumption to specific programming constructs, such as BBs, loops, and functions.

4.3. Analysis of Multithreaded Programs

In this article, we present the first steps toward energy consumption analysis of multithreaded programs. Two concurrency patterns are considered: replicated threads with no interthread communication, working on different sets of data (task farms), and pipelines of communicating threads. For both cases, we consider evenly distributed, balanced workloads.

There is a fundamental difference when statically predicting the case of interest (worst, best, average case) for time and for energy for multithreaded programs. Generally, for time only, the computations that contribute to the path forming the case of interest must be considered. For energy, all computations taking place during the case of interest must be considered. For instance, in an unbalanced task farm, the WCET will be equivalent to the longest running thread. To bound energy, the energy consumed by each thread needs to be aggregated. Thus, the static analysis needs to determine the number of active threads at each point in time to apply the energy model from Equation (1) and characterize the CFG of each thread. Then, IPET can be applied to each thread's CFG, extracting energy consumption bounds. Aggregating these together will give a loose upper bound on the program's energy consumption, meaning that the safety of the bound cannot be guaranteed.

In our balanced task farm examples, all task threads are active in parallel for the duration of the test. Thus, the number of active threads is constant, giving a constant N_t, used to determine the pipeline occupancy scaling factor, M, in Equation (1). For balanced pipelined programs, we consider the continuous, streaming data use case, so the same constant thread count property holds. In both cases, IPET can be performed on each thread's CFG and the results aggregated to retrieve the total energy consumption. The energy model covers the core-local instructions used for thread communication. Although off-core communication uses the same instructions as core-local communication, an additional energy cost needs to be accounted for external link usage. This cost was determined empirically for the development board used in Section 5.1.2.

For multithreaded programs with synchronous communications, to retrieve a WCET, IPET can be applied on a global graph, connecting the CFGs of all threads along communication edges. The communication edges can be treated by the IPET as normal CFG edges, and WCET can be extracted by solving the formulated problem [Potop-Butucaru and Puaut 2013]. This will return a single worst-case path across the global graph. Bounding energy in this way is not possible, as parallel thread activity over time needs to be considered. Activity can be blocked if the threads' workloads are unbalanced, due to the synchronous blocking message passing. In this case, statically determining the number of active threads at each point in time is hard.

Similar to SRA, profiling-based energy estimation does not need to infer the number of active threads at each execution stage for our balanced task farms and pipelined test cases. Therefore, retrieving the BB execution counters is sufficient.

The concurrency patterns addressed here represent typical embedded use cases. We demonstrate in Section 5.1.2 that for these use cases, SRA can provide sufficiently accurate information for design space exploration. More advanced concurrency analysis techniques, such as the ones employed in Shih et al. [2014], could be combined with our techniques to scale our analysis to more complex concurrency patterns.

5. EXPERIMENTAL EVALUATION

Two open source benchmark suites were used for evaluation. The first one, the `Mälardalen WCET` benchmark suite [Gustafsson et al. 2010], is specially designed for

WCET analysis. The second, the BEEBS benchmark suite [Pallister et al. 2013], is targeted at evaluating the energy consumption of embedded processors. The selected benchmarks were modified to work with our test harness and, in some cases, to make them more parametric to function input arguments. When necessary, benchmarks using floating point were modified to use integer values, as the Xcore does not support floating point operations. Some of the benchmarks were also extended to be multithreaded task farms, where the same code runs on two or four threads. Furthermore, a range of industrial benchmarks was selected to demonstrate the value of our estimation techniques to embedded developers.

Deeply embedded processors do not typically have hardware support for division or floating point operations, using software libraries instead. Software implementations are usually far less efficient than their hardware equivalent, both in terms of execution time and energy consumption. The effect of these software implementations on energy consumption should be known by developers, and therefore we include software division and software-emulated floating point arithmetic benchmarks. A radix-4 software divider, `Radix4Div` [Field 2014], is used. A less efficient version, `B.Radix4Div`, is added for comparison; it omits an early return when the dividend is greater than 255. As a consequence, CFG paths become more balanced, with less variation between the possible execution paths. The effect of this on the energy consumption is discussed later in this section. For software floating point, single-precision `SFloatAdd32bit` and `SFloatSub32bit` operations from Hauser [2014] are analyzed.

To extend our analysis to multithreaded communicating programs, we analyze two common signal processing tasks written for the Xcore: `FIR` and `Biquad` [XMOS 2014a]. Both tasks are implemented as pipelines of seven threads. Such programs are the preferred form for Xcore, as spreading the computation across threads allows the voltage and frequency of the core to be lowered, significantly reducing energy consumption while retaining the same performance as the single-threaded version.

A complete list of all benchmarks' attributes can be found in Georgiou [2016a]. Benchmarks were compiled with `xcc` version 12 [XMOS 2014b] at optimization level `O2`, which is the default for most compilers. For hardware measurements, as in Kerrison and Eder [2015], a shunt resistor current sense and sampling circuit obtains power dissipation with submilliwatt precision and variation of less than 1%.

5.1. SRA Results

For the evaluation of our SRA estimations together with the mapping technique, 20 benchmarks were used. These cover a broad spectrum of language features and code complexity. A combination of good understanding of the underlying algorithms, profiling information, and brute forcing of the benchmarks' functions input space was necessary to identify tests covering the algorithmic worst-case execution path for each benchmark with certainty.

Figure 4 presents the error margin of using the same ISA energy model with three energy estimation techniques compared to hardware energy measurements for our benchmarks. *Simulation* produces energy estimations based on instruction traces from ISS, *ISA SRA* uses the model for static analysis at the ISA level, and *LLVM IR SRA* uses our mapping technique to apply the model and analysis at LLVM IR level.

In the case of LLVM IR SRA, the accuracy of the prediction is heavily dependent on the accuracy of the mapping techniques presented in Section 3. As shown in Figure 4, for all benchmarks, the LLVM IR SRA results are within one percentage point error of ISA SRA results, except for the `Base64` and `levenshtein` benchmark with a further 4.3 and 2.0 percentage points error, respectively. In these cases, the CFGs of the two levels were significantly different due to new BBs introduced from branches in the ISA-level

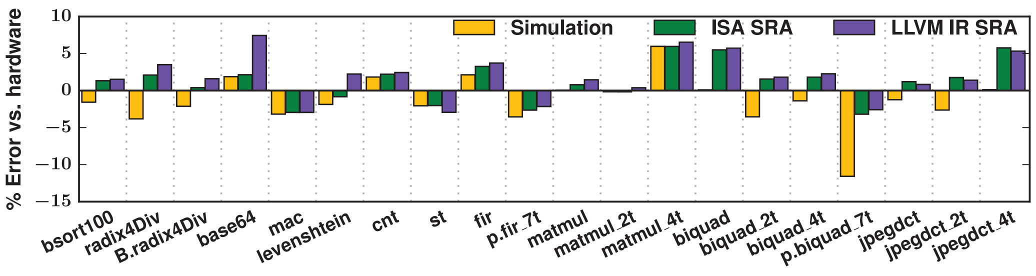


Fig. 4. SRA and ISS against hardware measurements (worst-case program inputs).

CFG. As discussed in Section 3.3, substantial differences between the two CFGs can reduce the effectiveness of our current tuning implementation.

Generally, for all results, a proportion of error is present in both forms of static analysis as well as simulation-based energy estimation. The error in the ISS-based estimation is a baseline for the best achievable accuracy in static analysis, as the ISS produces the most accurate execution statistics. For all benchmarks, the ISA SRA results are overapproximating the trace-based energy estimations. This applies also to the LLVM IR SRA results, with exception of the `st` benchmark. This overapproximation is a product of the bound analysis used, which is trying to select the most energy-costly CFG path based on the provided cost model.

The majority of SRA energy estimations are tightly overestimating the actual energy consumption measured on the hardware, up to 5.7% for ISA SRA and up to 7.4% for LLVM IR SRA. There are several cases for which the retrieved estimations underestimate the actual energy consumption (`mac`, `levenshtein`, `st`, `p.fir_7t`, `p.biquad_7t`). IPET is intended to provide bounds based on a given cost model. In our case, it tries to select the worst-case execution paths in terms of the energy consumption. However, energy consumption is data sensitive—that is, the energy cost of executing an instruction varies depending on (the circuit switching activity caused by) the operands used. Therefore, the observed underestimation is a consequence of using a data-insensitive energy model and analysis.

The SRA estimations seen in Figure 4 represent a loose upper bound on the benchmarks' energy consumption. These bounds cannot be considered safe for use in mission-critical applications. However, our experimental results show a low level of SRA underestimation (less than 4%), and therefore our SRA can still provide valuable guidance to the application programmer (e.g., to compare coding styles or algorithms).

Beyond the use of SRA predictions for bounding the energy consumption, several other use cases were investigated and are detailed next.

5.1.1. SRA Alternative Use Cases. The modified `B.Radix4Div` benchmark avoids an early return when the dividend is greater than 255. Omitting this optimization is less efficient but balances the CFG paths. The effect of this modification can be seen in Figure 5. The ISA-level energy consumption lower and upper bounds (the best and worst case retrieved by IPET) are shown. In the optimized version, `Radix4Div`, the energy consumption across different test cases varies significantly, creating a large range between the upper and lower energy consumption bounds. Conversely, the unoptimized version, `B.Radix4Div`, shows a lower variation, thus narrowing the margin between the upper and lower bounds, but has a higher average energy consumption.

Knowledge of such energy consumption behavior can be of value for applications like cryptography, where the power profile of systems can be monitored to reveal sensitive information in side channel attacks [Kocher et al. 1999]. In these situations, SRA can

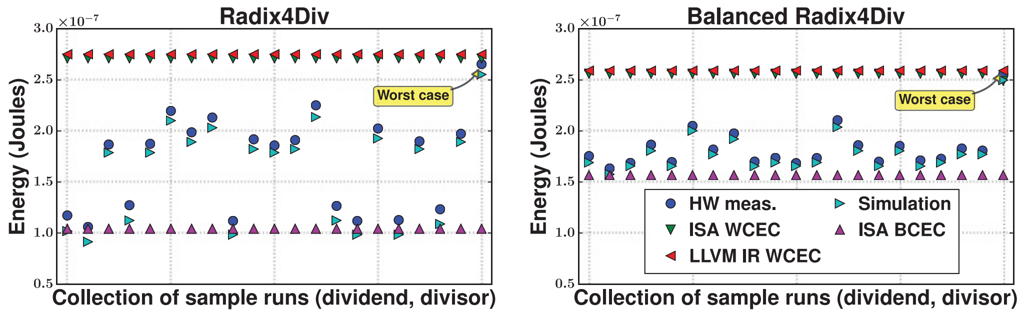


Fig. 5. Energy consumption bounds of the optimized and unoptimized version of `Radix4Div` benchmarks, captured by SRA.

Benchmark	ID	Configuration			
		C	T	V	F
MatMult (4 pairs of 30x30 matrices)	M1	1	1	1	450
	M2	1	2	1	450
	M3	1	4	1	450
Biquad Filter	B1	1	1	1	450
	B2	1	7	0.75	150
	B3	2	7	0.7	75

C: Number of cores used
T: Number of threads used
V: Core(s) Voltage in Volts
F: Core(s) Frequency in MHz

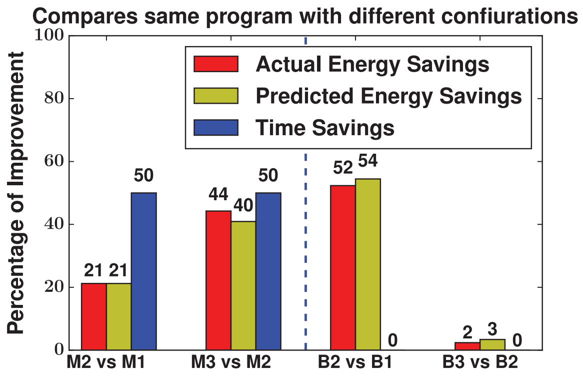


Fig. 6. Design space exploration enabled by SRA.

help developers design code with low energy consumption variation so that information that could be leaked through power monitoring can be obfuscated.

5.1.2. Design Space Exploration Using SRA. In this section, we examine how ISA SRA can be used as an alternative to measurements or simulation for design space exploration of task farms and pipelined programs. We consider applications for which the worst-case execution path is actually the dominant execution path. Having several available threads, several cores, and the ability to apply voltage and frequency scaling provides a wide range of configuration options in the design phase, with multiple optimization targets. This can include optimizing for quality of service, time, and energy, or a combination of all three. For this experiment, we used the XMOS XP-SKC-A16 [XMOS 2016b] development board, which consists of two interconnected Xcores with 16 threads available in total and supports voltage and frequency scaling.

The left-hand side of Figure 6 shows different configurations for the same program; on the right-hand side, these different versions are compared with regard to savings, if any, achieved for time and energy. For energy, both the hardware measurement–based and the SRA-predicted percentages of savings are shown to assess the ability of SRA to support sound energy-aware decisions. Our analysis can also infer the time figures; however, due to the time-deterministic nature of the architecture, the predictions closely match the actual values and therefore we omit them from the graph.

First, SRA was applied to replicated noncommunicating threads. The user can make energy-aware decisions on the number of threads to use with respect to time and energy estimations retrieved by our analysis. As an example, consider four independent matrix multiplications on four pairs of equally sized matrices (30×30). For all three versions, we keep the number of cores, the core voltage, and frequency constant, but we alter the number of threads and split the computation across them. The single-thread version, `M1`, has an execution time of $4\times$ the time needed to execute one matrix multiplication. However, the two-thread version, `M2`, halves the execution time and, as indicated by both SRA and actual measurements, decreases the energy by 21% compared to `M1`. The four-thread version, `M3`, halves the execution time again, and as predicted by SRA, the energy consumption is decreased by 40% compared to the two-thread version, `M2`. For the same case, `M3 vs M2`, the actual energy savings are 44%. Although there is a different estimation error between different numbers of active threads, the error range of 5% is small enough to allow comparisons between these different versions by only using the energy estimations from SRA. The measured and the predicted values are strongly correlated and confirm that using more threads increases the power dissipation, but the reduction in execution time saves energy on the investigated platform.

Next, SRA was applied to streaming pipelines of communicating threads. There is a choice in how to spread the computation across threads to maximize throughput to either minimize execution time or lower the necessary device operating frequency and voltage while maintaining performance. Our SRA can take advantage of the fact that the energy model used is parametric to voltage and frequency, to statically identify the most energy-efficient configuration of the same program, among several options that deliver the same throughput. We demonstrate this on the `Biquad` benchmark.

The `Biquad` benchmark implements an equalizer; it takes a signal and attenuates or amplifies different frequency bands. The `Biquad` equalizer uses a cascade of biquad filters. Each biquad filter attenuates or amplifies one specific frequency range; the signal is sequentially passed through all filters, eventually attenuating or amplifying all bands. In a standard equalizer setting, a seven-bank filter is used, of which the low four banks are set to amplify and the top three banks are set to attenuate. These seven banks are implemented in a seven-stage pipeline.

Figure 6, on the left-hand side, shows the three different versions explored for the `Biquad` filter. `B1` is the sequential version running at 450MHz and 1V. The single-core parallel version, `B2`, uses seven threads, one for each bank, and runs at 150MHz and 0.75V. The multicore parallel version, again using seven threads, places four threads on the first core and three on the second core. It therefore allows for halving the core frequency to 75MHz and a further reduction of the core voltage to 0.7V. All versions, `B1, B2, and B3,` deliver the same filter performance, and therefore the percentage of time savings on the right-hand side graph of Figure 6 is zero. In contrast, we can see a predicted energy savings on `B2 vs B1` of 54% and on `B3 vs B2` of 3%. The actual energy savings are 52% and 2%, respectively. The small error of our energy consumption estimations enables energy-aware decisions just by the use of SRA. In this case, version `B2` is the best solution, since while maintaining the same performance, it halves the energy consumption. In comparison, the small additional energy savings realized by `B3` may not warrant the use of a second core. These energy-aware decisions cannot be achieved by only examining the execution time of each one of the three configurations. Similar results were achieved for the `Fir` benchmark.

In both of the preceding examples, we demonstrated that our SRA is sufficient to provide design space exploration guidance considering both time and energy, without the need for any simulation or hardware measurements. This has significant benefits since SRA is faster than simulation and less costly to deploy than hardware measurements.

5.2. Profiling-Based Analysis Results

To evaluate the profiling-based energy estimation technique together with the mapping technique, 30 benchmarks were used. For these benchmarks, Figure 7 presents the

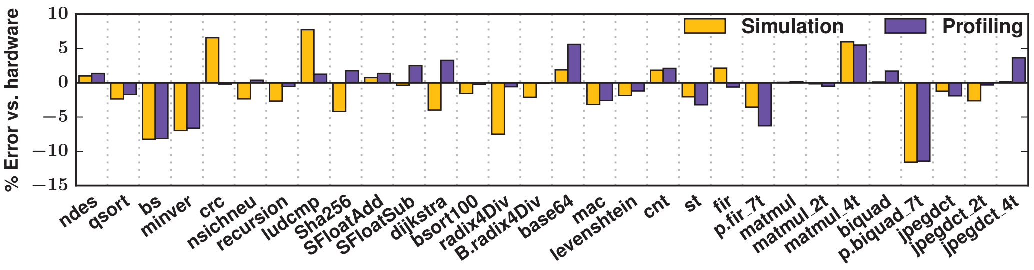


Fig. 7. Profiling- and ISS-based energy estimations against hardware measurements.

error margin of using the same ISA energy model with the ISS-based estimations and the profiling-based estimations compared to hardware energy measurements, respectively. The average absolute error obtained for the profiling-based estimations is 3.1%, and 2.7% for the ISS-based estimations. The ISS-based estimation is more reliable than profiling. This is because cycle-accurate ISS allows for a very precise energy consumption estimation for each particular program execution as the program is executed in simulation with a specific set of input parameters. The exact sequence of instructions can be recorded during simulation and then used to estimate energy consumption. In contrast, profiling is less precise as it operates further away from the hardware, at the LLVM IR level. Profiling estimation precision heavily depends on the quality of the mapping techniques introduced in Section 3.2, and on the quality of the profiling techniques used to collect execution counts of LLVM IR BBs, as described in Section 4.2. However, our profiling results demonstrate a high accuracy with an average error deviation of 1.8% from the ISS.

5.2.1. Profiling-Based Estimation Performance. ISS execution is often several orders of magnitude slower than hardware. The performance of an ISS is governed by the complexity of the program's underlying algorithms. In contrast, the performance of our profiling is mainly governed by the program size, as retrieving the BB execution counts incurs a negligible execution time overhead when running on the actual platform, as discussed in Section 4.2. The bigger the program, the more time will be needed for Stage 1, `LLVM Instrumentation Pass`, in the `Profiling` phase, and Stage 1, `BB Energy Characterization`, in the `Energy Consumption Estimation` phase, shown in Figure 3.

In the case of benchmarks with low algorithmic complexity and function inputs that will trigger a very short simulation time, no significant performance gains will be observed from using profiling over ISS estimation. For example, for our `SFloatAdd` benchmark, with a $O(1)$ complexity, the average profiling estimation performance observed has a negligible gain over the ISS estimation. However, increasing the complexity of the benchmarks' underlying algorithms and using parameters that trigger longer simulation time results in the profiler significantly outperforming the simulation. For example, when using 30×30 size matrices for our matrix multiplication benchmark, the profiling estimation achieves a 381× speedup over the ISS estimation. The benchmark's small size allowed for a fast profiling estimation, but the ISS estimation performance follows the $O(n^3)$ algorithmic complexity of the benchmark.

A further speedup can be achieved for the profiling-based method when testing the same program with different inputs. If the optimized LLVM IR code obtained is the same across the different inputs, then there is no need to repeat the LLVM IR energy characterization stage. In such cases, using profiling energy estimation has clear performance advantages compared to ISS-based energy estimation.

5.3. Discussion

Our SRA-based estimation at the LLVM IR level has a deviation in the range of 1% from the ISA SRA, and our profiling-based estimation has an average error deviation of 1.8% from the ISS. This shows that the tuning phase introduced in Section 3.2.3 mitigates the impact of the mapping limitations described in Section 3.3 and yields sufficiently accurate results for the architecture and compiler under consideration. Depending on the architecture and the compiler implementation, the tuning phase heuristics can be further improved to achieve the required level of accuracy.

6. RELATED WORK

SRA is a methodology to determine usage bounds of a resource (usually time or energy or both) for a specific task when executed on a piece of hardware without actually executing the task. This requires accurate modeling of the hardware to capture the dynamic functional and nonfunctional properties of task execution. Determining these properties accurately is known to be undecidable in general. Therefore, to extract safe values for the resource usage of a task, a sound approximation is needed [Wilhelm et al. 2008; Brat et al. 2014].

SRA has been mainly driven by the timing analysis community. Static cost analysis techniques based on setting up and solving recurrence equations date back to the seminal work of Wegbreit [1975] and have been developed significantly in subsequent work [Rosendahl 1989; Debray et al. 1997; Vasconcelos and Hammond 2003; Navas et al. 2007; Albert et al.2011]. Other classes of approaches to cost analysis use dependent types [Hoffmann et al. 2012], SMT solvers [Alonso-Blas and Genaim 2012], or size change abstraction [Zuleger et al. 2012]. Generally, for performing an accurate WCET static analysis, there are four essential components [Wilhelm et al. 2008]:

(1) *Value analysis*, which is mainly used to analyze the behavior of the data cache.
(2) *Control flow analysis*, which is used to identify the dynamic behavior of a program.
(3) *Low-level analysis*, which attempts to retrieve timing costs for each atomic unit on a given hardware platform, such as an instruction or a BB in a CFG for a processor.
(4) *Calculation*, which uses the results from the two previous components to estimate the WCET. The most common techniques used for calculation of the WCET are the IPET, the path-based techniques, and the tree-based methods [Engblom et al. 2000].

Three of the preceding components, namely the control flow analysis, low-level analysis, and calculation, are adopted in our work, as discussed in Section 4.1.

IPET is one of the most popular methods used for WCET analysis [Li and Malik 1995; Theiling and Ferdinand 1998; Ottosson and Sjodin 1997; Engblom et al. 2000; Potop-Butucaru and Puaut 2013]. In this approach, the CFG of a program is expressed as an ILP system, where the objective function represents the program's execution time. The problem then becomes a search for the WCET by maximizing the retrieved objective function under some constraints on the execution counts of the CFG's BBs.

Although significant research has been conducted in static analysis for the execution time estimation of a program, there is little on energy consumption. One of the few approaches [Navas et al. 2008] seeks to statically infer the energy consumption of Java programs as functions of input data sizes by specializing a generic resource analyzer [Navas et al. 2007; Hermenegildo et al. 2005] to Java bytecode analysis [Navas et al. 2009]. However, a comparison of the results with actual measurements was not performed. Later, in Liqat et al. [2014], the same generic resource analyzer was instantiated to perform energy analysis of XC programs [Watt 2009] at the ISA level based on ISA-level energy models and including a comparison with actual hardware measurements. However, the scope of this particular analysis approach was limited to a small set of simple benchmarks because information required for the analysis of more complex programs, such as program structure and types, is not available at the ISA level. The SRA presented in this article does not rely on such information. A similar approach, using cost functions, was used in Grech et al. [2015]. The analysis was performed at the LLVM IR level using an early version of the mapping technique that we formalize and describe in full detail for the first time in this work. Although the range of programs that could be analyzed was improved, compared to Liqat et al. [2014], the complexity of solving recurrence equations for analyzing larger programs proved a limiting factor.

In Jayaseelan et al. [2006], the WCEC for a program was inferred by using the IPET first introduced in Li and Malik [1995]. Although the authors manage to bound the WCEC on a simulated processor by maximizing the switching activity factor for each simulated component, they acknowledge the need to validate their estimation results against commercial embedded processors. Similarly, in Wägemann et al. [2015], the authors attempt to perform static WCEC analysis for a simple embedded processor, the `ARM Cortex M0+`. This analysis is also based on IPET combined with a so-called absolute energy model, an energy model that is said to provide the "maximum energy consumption of each instruction." The authors argue that they can retrieve a safe bound. However, this is demonstrated on a single benchmark, `bubblesort`, only. They also acknowledge that using an absolute energy model can lead to significant overestimation, making the bounds less useful. We choose to use a pseudorandom data characterized energy model, as empirical evidence shows that such models tend to be close to the actual worst case [Pallister et al. 2015b].

All of the reviewed previous works combined SRA techniques together with energy models to capture the WCEC path. Currently, there is no practical method to perform average case static analysis [Townley 2013]. One of the most recent works toward average case SRA [Schellekens 2010] demonstrates that compositionality combined with the capacity for tracking data distributions unlocks the average case analysis, but novel language features and hardware designs are required to support these properties. This situation motivated us to develop a dynamic profiling technique that captures the actual energy consumption of a program based on its input parameters.

There are two main requirements for CPU energy profiling. First, a technique is needed to instrument a program's execution and collect data that represent the program's dynamic behavior. Code instrumentation, simulation, and hardware performance counters are some of the most popular techniques used to collect such data. The second requirement is a method of associating energy information with the various entities, events in the set of the collected instrumentation data. This is typically done either through an energy model or by power monitoring the program's execution.

Energy modeling and profiling can be performed at various levels of abstraction. In Bogliolo et al. [1997], a gate-level power consumption model was created and combined with event-driven logic simulation to estimate the power consumption of programs. Modeling and profiling at such low design levels is not practical for most commercial embedded processors, as their lower-level circuit information is not available. Moreover, this estimation is quite slow and not practical for fine-grain energy characterization of software.

In Tiwari et al. [1996], an ISA-level energy model was proposed that treated the hardware as a black box and obtained energy consumption data through hardware measurements of large loops of individual instructions. Modeling at the ISA allowed for attributing energy costs to low-level software components such as ISA BBs. This led to further research that combined ISA energy models with instruction set simulators and profilers to extract energy estimations [Sarta et al. 1999; Brooks et al. 2000; Steinke et al. 2001; Sami et al. 2002; Ibrahim et al. 2008]. Our energy model is also based on Tiwari et al. [1996] but is significantly extended to a multithreaded version for the Xcore, which takes into account both interinstruction and interthread effects.

For more complex processors or for system-level energy consumption estimation, energy modeling and profiling at the ISA level is impractical. In such cases, performance counter–based statistical energy modeling and estimation is preferable. In several works [Contreras and Martonosi 2005; Shao and Brooks 2013; Schubert et al. 2012], the authors used performance counters to characterize the processor energy consumption based on the conditions affecting these counters, such as cache misses and prefetches. Then, by combining this energy characterization with performance counters' execution statistics, they predict the energy consumption of an application. An alternative to performance counters is the use of an external multimeter to directly measure a system's energy profile [Weaver et al. 2012].

In Brandolese et al. [2011], energy characterization of LLVM IR code is performed by linear regression analysis. This is combined with instrumentation and execution on a target host machine to estimate the performance and energy requirements in embedded software. Transferring the LLVM IR energy model to a new platform requires performing the regression analysis again. The mapping technique that we present here does not involve any regression analysis, and it is fully portable. It requires only the adjustment of the LLVM mapping pass to the new architecture. Furthermore, our LLVM IR mapping technique provides an on-the-fly energy characterization that takes into consideration the compiler behavior including optimization and transformation passes and other architecture-specific aspects. The instrumentation technique of Brandolese et al. [2011] is also based on collecting execution BB counts, but this is done by executing programs on a host machine rather than the actual platform. However, compiling and executing on a higher-end PC rather than the target deeply embedded device could yield an execution profile significantly different from the one that would be obtained on the target machine. Our profiling technique collects execution counts on the target machine and maps them back to their corresponding LLVM IR BB.

In Ozturk et al. [2013], a scheme is proposed that enables the compiler to exploit both task and data parallelism and automatically map an application to an embedded multiprocessor containing voltage islands. The scheme is based on the workload of each processor using both hardware parallelism and voltage scaling to reduce energy consumption without increasing the overall execution time. Per-core frequency and voltage configurations can be provided to appropriately design Xcore-based systems. Using our profiling technique and our LLVM IR energy characterization, accurate workload estimations can be achieved at the LLVM IR level. These could be utilized by the optimization scheme introduced in Ozturk et al. [2013] to enable the LLVM compiler to achieve similar energy consumption savings.

Profiling has previously been used successfully to enable energy-aware compilation. A combination of code analysis and profiling techniques can enable the compiler to build power-aware flow graphs [Rele et al. 2002]. Such graphs have their BBs annotated with the hardware resource requirements and the execution counts. The power-aware flow graphs can then be used to identify code regions where several of the processor's functional units can be turned off during execution to reduce the static power dissipation. Lin et al. [2015] demonstrate a framework that enables the compiler to support several energy-related optimizations for parallel design patterns. A series of pragmas were introduced to identify specific parallel design patterns and guide the compiler to apply power optimizations. These optimizations were enabled by utilizing power profiling and code instrumentation feedback provided by a simulator.

Our profiler provides energy consumption information directly into the compiler's optimizer. Therefore, it can enable feedback-directed compilation that targets energy-specific optimizations and design space exploration for deeply embedded systems.

7. CONCLUSION AND FUTURE WORK

This work focuses on providing energy consumption estimation techniques that can enable energy transparency in the software stack of deeply embedded processors typically used in IoT applications. A novel target-agnostic mapping technique is introduced to allow energy characterization of the LLVM tool chain IR, namely the LLVM IR. This is a significant step beyond existing ISA-level energy estimation techniques. Our mapping technique can give powerful insights to the LLVM IR optimizer regarding execution time and the energy consumption of a program. This enables programmers to investigate how optimizations can affect their program's energy consumption, or even help introduce new energy-specific optimizations in the future.

For energy consumption bounds, an IPET-based SRA was developed. The analysis was introduced at both ISA and LLVM IR levels. The results demonstrated that the mapping technique enables SRA at the LLVM IR level with a small accuracy loss in the range of only 1% compared to SRA at the ISA level. SRA-based energy estimation was also applied to a set of multithreaded programs for the first time to the best of our knowledge. This is a significant step beyond existing work that examines single-thread programs. As shown in Section 5.1.2, such an analysis can provide significant guidance for time-energy design space exploration between different numbers of threads and cores.

To estimate the actual energy consumption of a program under specific input parameters, a target-agnostic profiling technique was developed. The technique is enabled by the LLVM IR energy characterization utilizing our mapping. It is designed to ensure that the instrumentation code required for profiling does not lead to energy overheads. Experimental evaluation shows an average absolute error of 3.2% compared to hardware measurements. Our profiling technique is more flexible and significantly more efficient than ISS-based estimation. It can attribute energy consumption to software components, such as BBs and functions, in contrast to hardware measurements. The high accuracy and performance of our profiler can enable feedback-directed optimization for energy consumption. LLVM IR energy consumption estimations of each compilation can be taken into consideration iteratively in subsequent compilations. Each new compilation will be able to make more energy-aware decisions.

The techniques introduced in this article are focused on deeply embedded architectures that favor predictability over performance. Such architectures are the backbone of IoT applications. Enabling energy-aware software development for such architectures will help to tackle the energy challenge that IoT faces. We acknowledge that some of our techniques, mainly SRA-based estimation, work best with predictable architectures. This is also true for WCET analysis. In fact, static analysis is inherently limited in that sense. Thus, in future work, we intend to examine the portability of our profiling-based techniques to more complex processors.

Future work aims to analyze more complex concurrent programs, such as distinct noncommunicating threads or pipelines of threads with unbalanced workloads. We anticipate that our LLVM IR profiling technique will scale better to these cases than SRA. As the core count of the analyzed system grows, more in-depth exploration of energy saving techniques such as per-core DVFS could be considered both at the model level and SRA or profiling levels.

REFERENCES

E. Albert, P. Arenas, S. Genaim, and G. Puebla. 2011. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning* 46, 2, 161–203.

D. E. Alonso-Blas and S. Genaim. 2012. On the limits of the classical approach to cost analysis. 7460, 405–421. DOI:http://dx.doi.org/10.1007/978-3-642-33125-1_27

ARM. 2016. Cortex-M Series Family. Retrieved February 17, 2017, from http://www.arm.com/products/processors/cortex-m.

C. Blackmore, O. Ray, and K. Eder. 2015. A logic programming approach to predict effective compiler settings for embedded software. *Theory and Practice of Logic Programming* 15, 4–5, 481–494. DOI:http://dx.doi.org/10.1017/S1471068415000174

A. Bogliolo, L. Benini, G. D. Micheli, and B. Ricc. 1997. Gate-level power and current simulation of CMOS integrated circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 5, 4, 473–488. DOI:http://dx.doi.org/10.1109/43.736184

C. Brandolese, S. Corbetta, and W. Fornaciari. 2011. Software energy estimation based on statistical characterization of intermediate compilation code. In *Proceedings of the 2011 International Symposium on Low Power Electronics and Design (ISLPED'11)*. 333–338. DOI:http://dx.doi.org/10.1109/ISLPED.2011.5993659

G. Brat, J. A. Navas, N. Shi, and A. Venet. 2014. IKOS: A framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods*. Springer, 271–277.

D. Brooks, V. Tiwari, and M. Martonosi. 2000. Wattch: A framework for architectural-level power analysis and optimizations. *ACM SIGARCH Computer Architecture News* 28, 2, 83–94. DOI:http://dx.doi.org/10.1145/342001.339657

G. Contreras and M. Martonosi. 2005. Power prediction for Intel XScale processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED'05)*. IEEE, Los Alamitos, CA, 221–226. DOI:http://dx.doi.org/10.1109/LPE.2005.195518

S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. 1997. Lower bound cost estimation for logic programs. In *Proceedings of the 1997 International Logic Programming Symposium*. 291–305.

DWARF. 2013. The DWARF Debugging Standard. Retrieved February 17, 2017, from http://dwarfstd.org/.

K. Eder, J. P. Gallagher, P. López-García, H. Muller, Z. Banković, K. Georgiou, R. Haemmerlé, et al. 2016. ENTRA: Whole-systems energy transparency. *Microprocessors and Microsystems*, 278–286. DOI:http://dx.doi.org/10.1016/j.micpro.2016.07.003

K. Eder, K. Georgiou, and N. Grech (Eds.). 2013. *Common Assertion Language*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337). Deliverable 2.1. Available at http://entraproject.eu.

J. Engblom, A. Ermedahl, and F. Stappert. 2000. Comparing different worst-case execution time analysis methods. In *Proceedings of the Work-in-Progress Session of the 21st IEEE Real-Time Systems Symposium (RTSS'00)*.

M. Field. 2014. Binary division. Retrieved February 17, 2017, from https://github.com/kg8280/EnergyTransparency/blob/master/Benchmarks/radix4Division.zip.

K. Georgiou. 2016a. Energy Transparency for Deeply Embedded Programs—Benchmarks. Retrieved February 17, 2017, from https://github.com/kg8280/EnergyTransparency.

K. Georgiou. 2016b. Energy Transparency for Deeply Embedded Programs—FNOP Modeling. Retrieved February 17, 2017, from https://github.com/kg8280/EnergyTransparency/blob/master/FnopModeling.pdf.

N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. 2015. 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)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/2764967.2764974

J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. 2010. The Mälardalen WCET benchmarks—past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*.

M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. 2005. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming* 58, 1–2, 115–140.

J. Hoffmann, K. Aehlig, and M. Hofmann. 2012. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems* 34, 3, 14.

S. J. Hollis and S. Kerrison. 2016. Swallow: Building an energy-transparent many-core embedded real-time system. In *Proceedings of the 2016 Design, Automation, and Test in Europe Conference and Exhibition (DATE'16)*. 73–78.

J. Hauser. 2014. Berkeley SoftFloat. Retrieved February 17, 2017, from http://www.jhauser.us/arithmetic/SoftFloat.html.

M. E. A. Ibrahim, M. Rupp, and H. A. H. Fahmy. 2008. Power estimation methodology for VLIW digital signal processors. In *Proceedings of the 2008 42nd Asilomar Conference on Signals, Systems, and Computers*. IEEE, Los Alamitos, CA, 1840–1844. DOI:http://dx.doi.org/10.1109/ACSSC.2008.5074746

R. Jayaseelan, T. Mitra, and X. Li. 2006. Estimating the worst-case energy consumption of embedded software. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*. 81–90. DOI:http://dx.doi.org/10.1109/RTAS.2006.17

S. Kerrison and K. Eder. 2015. Energy modeling of software for a hardware multithreaded embedded microprocessor. *ACM Transactions on Embedded Computing Systems* 14, 3, 56:1–56:25. DOI:http://dx.doi.org/10.1145/2700104

P. C. Kocher, J. Jaffe, and B. Jun. 1999. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'99)*. 388–397.

C. Lattner and V. S. Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'04)*. 75–88.

C. Lattner and D. Patel. 2014. Extensible Metadata in LLVM IR. Retrieved February 17, 2017, from http://blog.llvm.org/2010/04/extensible-metadata-in-llvm-ir.html.

Y.-T. S. Li and S. Malik. 1995. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference (DAC'95)*. ACM, New York, NY, 456–461. DOI:http://dx.doi.org/10.1145/217474.217570

Y.-T. S. Li and S. Malik. 1997. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16, 12, 1477–1487. DOI:http://dx.doi.org/10.1109/43.664229

C.-Y. Lin, C.-B. Kuan, W.-L. Shih, and J. K. Lee. 2015. Compilers for low power with design patterns on embedded multicore systems. *Journal of Signal Processing Systems* 80, 3, 277–293. DOI:http://dx.doi.org/10.1007/s11265-014-0917-9

U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo, and K. Eder. 2014. 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)*.

LLVMorg. 2014. The LLVM Compiler Infrastructure. Retrieved February 17, 2017, from http://www.llvm.org/.

D. May. 2009. *The XMOS XS1 Architecture*. Retrieved February 17, 2017, from https://www.xmos.com/download/private/The-XMOS-XS1-Architecture%281.0%29.pdf.

J. Navas, M. Méndez-Lojo, and M. Hermenegildo. 2008. Safe upper-bounds inference of energy consumption for Java bytecode applications. In *Proceedings of the 6th NASA Langley Formal Methods Workshop (LFM'08)*.

J. Navas, M. Méndez-Lojo, and M. Hermenegildo. 2009. User-definable resource usage bounds analysis for Java bytecode. *Electronic Notes in Theoretical Computer Science* 253, 5, 65–82.

J. Navas, E. Mera, P. López-García, and M. Hermenegildo. 2007. User-definable resource bounds analysis for logic programs. In *Logic Programs*. Lecture Notes in Computer Science, Vol. 4670. Springer, 348–363.

G. Ottosson and M. Sjodin. 1997. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*. 47–55.

O. Ozturk, K. Mahmut, and C. Guangyu. 2013. Compiler-directed energy reduction using dynamic voltage scaling and voltage islands for embedded systems. *IEEE Transactions on Computers* 62, 2, 268–278. DOI:http://dx.doi.org/10.1109/TC.2011.229

J. Pallister, J. Bennett, and S. Hollis. 2013. The BEEBS Benchmark Suite. Retrieved February 17, 2017, from http://www.cs.bris.ac.uk/Research/Micro/beebs.jsp.

J. Pallister, K. Eder, and S. J. Hollis. 2015a. Optimizing the flash-RAM energy trade-off in deeply embedded systems. In *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*. 115–124. DOI:http://dx.doi.org/10.1109/CGO.2015.7054192

J. Pallister, K. Eder, S. J. Hollis, and J. Bennett. 2014. 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)*. ACM, New York, NY, Article No. 20. DOI:http://dx.doi.org/10.1145/2656106.2656108

J. Pallister, S. Kerrison, J. Morse, and K. Eder. 2015b. Data dependent energy modelling: A worst case perspective. arXiv:1505.03374.

D. Potop-Butucaru and I. Puaut. 2013. Integrated worst-case execution time estimation of multicore applications. In *Proceedings of the 13th International Workshop on Worst-Case Execution Time Analysis*. 21–31. DOI:http://dx.doi.org/10.4230/OASIcs.WCET.2013.21

S. Rele, S. Pande, S. Onder, and R. Gupta. 2002. *Optimizing Static Power Dissipation by Functional Units in Superscalar Processors*. Springer, Berlin, Germany, 261–275. DOI:http://dx.doi.org/10.1007/3-540-45937-5_19

M. Rosendahl. 1989. Automatic complexity analysis. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*. ACM, New York, NY, 144–156. DOI:http://dx.doi.org/10.1145/99370.99381

M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria. 2002. An instruction-level energy model for embedded VLIW architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21, 9, 998–1010. DOI:http://dx.doi.org/10.1109/TCAD.2002.801105

D. Sarta, D. Trifone, and G. Ascia. 1999. A data dependent approach to instruction level power estimation. In *Proceedings of the 1999 IEEE Alessandro Volta Memorial Workshop on Low-Power Design*. 182–190. DOI:http://dx.doi.org/10.1109/LPD.1999.750419

M. P. Schellekens. 2010. MOQA: Unlocking the potential of compositional static average-case analysis. *Journal of Logic and Algebraic Programming* 79, 1, 61–83. DOI:http://dx.doi.org/10.1016/j.jlap.2009.02.006

S. Schubert, D. Kostic, W. Zwaenepoel, and K. G. Shin. 2012. Profiling software for energy consumption. In *Proceedings of the 2012 IEEE International Conference on Green Computing and Communications (GreenCom'12)*. 515–522. DOI:http://dx.doi.org/10.1109/GreenCom.2012.86

Y. S. Shao and D. Brooks. 2013. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'13)*. IEEE, Los Alamitos, CA, 389–394. DOI:http://dx.doi.org/10.1109/ISLPED.2013.6629328

W. Shih, Y.-P. You, C.-W. Huang, and J. K. Lee. 2014. Compiler optimization for reducing leakage power in multithread BSP programs. *ACM Transactions on Design Automation of Electronic Systems* 20, 1, Article No. 9. DOI:http://dx.doi.org/10.1145/2668119

A. Srivastava and A. Eustace. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI'94)*. ACM, New York, NY, 196–205. DOI:http://dx.doi.org/10.1145/178243.178260

S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel. 2001. An accurate and fine grain instruction-level energy model supporting software optimizations. In *Proc.eedings of the 2001 International Workshop on Power and Timing Modeling, Optimization, and Simulation (PATMOS'01)*. DOI:http://dx.doi.org/10.1.1.115.3528

H. Theiling and C. Ferdinand. 1998. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*. 144–153. DOI:http://dx.doi.org/10.1109/REAL.1998.739739

V. Tiwari, S. Malik, A. Wolfe, and M. Tien-Chien Lee. 1996. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 13, 2–3, 223–238. DOI:http://dx.doi.org/10.1007/BF01130407

J. M. Townley. 2013. *Practical Programming for Static Average-Case Analysis: The MOQA Investigation*. Ph.D. Dissertation. University College Cork.

P. Vasconcelos and K. Hammond. 2003. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Implementation of Functional Languages*. Lecture Notes in Computer Science, Vol. 3145. Springer, 86–101.

P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza, and W. Schröder-Preikschat. 2015. Worst-case energy consumption analysis for energy-constrained embedded systems. In *Proceedings of the 2015 27th Euromicro Conference on Real-Time Systems (ECRTS'15)*.

D. Watt. 2009. *Programming XC on XMOS Devices*. XMOS Limited. http://books.google.co.uk/books?id=81klKQEACAAJ.

V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. 2012. Measuring energy and power with PAPI. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops (ICPPW'12)*. IEEE, Los Alamitos, CA, 262–268. DOI:http://dx.doi.org/10.1109/ICPPW.2012.39

B. Wegbreit. 1975. Mechanical program analysis. *Communications of the ACM* 18, 9, 528–539.

T. Wei, J. Mao, W. Zou, and Y. Chen. 2007. A new algorithm for identifying loops in decompilation. In *Static Analysis*. Lecture Notes in Computer Science, Vol. 4634. Springer, 170–183. DOI:http://dx.doi.org/10.1007/978-3-540-74061-2_11

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, et al. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems* 7, 3, Article No. 36. DOI:http://dx.doi.org/10.1145/1347375.1347389

XMOS. 2014a. Code to perform standard DSP functions, such as Biquads, FIRs, sample rate conversion. Retrieved February 17, 2017, from https://github.com/xcore/sc_dsp_filters.

XMOS. 2014b. xTIMEcomposer. Retrieved February 17, 2017, from https://www.xmos.com/support/xtools.

XMOS. 2016a. Debug with Printf in Real-Time. Retrieved February 17, 2017, from https://www.xmos.com/support/tools/other?subcategory=&component=14774.

XMOS. 2016b. xCORE-Analog SliceKIT. Retrieved February 17, 2017, from https://www.xmos.com/support/boards?product=17554.

F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. 2012. Bound analysis of imperative programs with the size-change abstraction (extended version). arXiv:1203.5303.

Received September 2016; revised December 2016; accepted January 2017