Taskify: An Integrated Development Environment to Develop and Debug Intermittent Software for the Batteryless Internet of Things

Murat Mülayim*, Arda Goknil† and Kasım Sinan Yıldırım*‡
*Department of Computer Engineering, Ege University, Turkey
†SnT, University of Luxembourg, Luxembourg
‡Department of Information Engineering and Computer Science, University of Trento, Italy
Email: *murat.mulayim74@gmail.com, †ar.goknil@gmail.com, ‡kasimsinan.yildirim@unitn.it

***Abstract*—Batteryless embedded devices rely only on ambient energy harvesting that enables stand-alone and sustainable applications for the Internet of Things. These devices perform computation, sensing, and communication when the harvested ambient energy in their energy reservoir is sufficient; they die abruptly when the energy drains out completely. This kind of operation, the so-called intermittent execution, dictates a task-based programming model for the development and implementation of intermittent applications. However, today's task-based intermittent programs are tightly-coupled to the underlying run-time environments. This makes their debugging and testing difficult before deploying them into the target platform. To remedy this, we present Taskify, a tool that enables engineers to write and debug task-based intermittent programs in TaskDSL, i.e., a domain-specific language we designed for the development of intermittent programs on any general-purpose computer. Taskify automatically transforms these programs into C programs that can be linked to the underlying run-time environment and deployed into the target platform. Taskify is implemented as an Eclipse plugin. It has been evaluated on three intermittent applications.**

***Index Terms*—Energy Harvesting, Intermittent Software, Batteryless, Tool, Debugger**

I. Introduction

Radio Frequency (RF)-powered computers emerged with the recent advancements in RF harvesting circuits and micro-controllers having ultra-low power requirements [1], [2]. These computers operate without batteries and enable stand-alone and sustainable applications for the Internet of Things (IoT) [1]–[3]. In particular, embedding IoT devices in the human body, in everyday items like clothes or inhospitable locations are now feasible with batteryless operation [4].

A typical RF-powered computer, e.g. WISP [5], is equipped with an ultra-low-power micro-controller, several sensors, a non-volatile secondary memory (FRAM [6]), and a capacitor for energy storage. These computers sense, compute and communicate when the harvested energy in the capacitor is sufficient; they die abruptly when the capacitor drains out completely. Due to unpredictable ambient energy sources [7], power failures may reach a rate of 10 times per second [8], [9]. Upon power failure, the *volatile state*, i.e., the general and special purpose registers of the micro-controller and the content

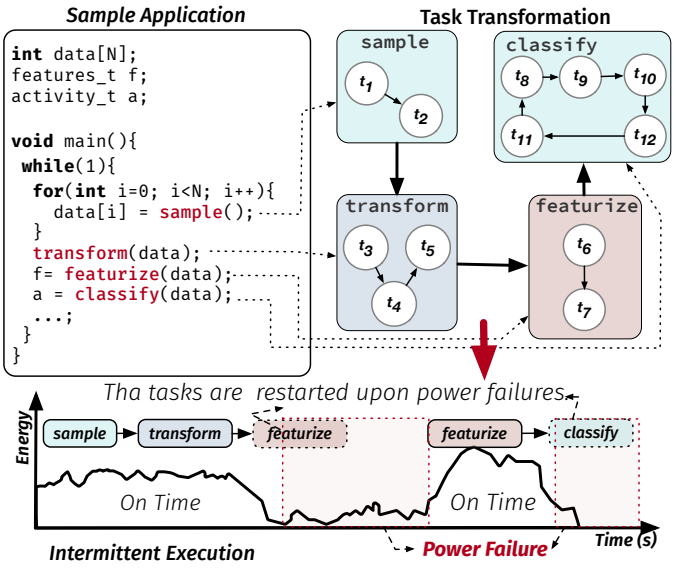

Fig. 1. Software designed for continuously-powered systems cannot progress on intermittent computers. Engineers need to split the computation into a set of tasks that can run within the energy stored in the energy reservoir. The tasks have all-or-nothing semantics and they can be restarted upon power failures without any side effects.

of the volatile memory, is lost but the *non-volatile state*, i.e., the content of FRAM, persists. Frequent power failures lead to an inherently *intermittent execution* for the software running on these computers. However, during intermittent execution of programs, the following problems arise:

- the progress of computation *might not be guaranteed* due to the frequent loss of the volatile state [10];
- the re-execution of the software upon recovery from a power failure might lead to either *semantically incorrect results* due to the corrupted non-volatile state [11] or to a *system crash* [12].

There is an ample body of recent work addressing these problems. They can be investigated under two categories: *checkpointing-based* and *task-based systems*. *Checkpointing*-based systems [8], [13]–[17] simply log the *volatile state* of

the computation in persistent memory by placing checkpoints. Upon power failure, the computation continues from the latest checkpoint. However, checkpoints introduce non-negligible overhead since their time and energy cost grow with the size of the volatile state.

Task-based systems [12], [18]–[20] introduce considerably less overhead by introducing a static *task model* for programming intermittent computers (see Fig. 1). These systems enable engineers to decompose their programs into a collection of tasks at compile time. Engineers also identify the control-flow and how the data is shared and manipulated by the tasks. The tasks have atomic all-or-nothing semantics so that their re-execution does not leave non-volatile memory in an inconsistent state—task-based systems ensure the atomic completion of the tasks despite arbitrary power failures.

A. Problem Statement

Ensuring the functional correctness of task-based applications, and in turn debugging intermittent systems, is a burden for engineers. This is due to the following reasons:

- Splitting the computation into several tasks and identifying the control-flow and data sharing/communication among the tasks require a significant developer effort [17]. This procedure is also error-prone. On the other hand, it is time-consuming to verify if the task splitting is done correctly and if the task-based program is functionally correct.
- Task-based intermittent programs are written by using the programming constructs provided by the hardware-dependent run-time environments [12], [18]–[20]. Therefore, they are tightly coupled to the underlying environment, and engineers can debug them only after deploying them into the target hardware platform. To the best of our knowledge, there is no tool support for debugging and testing these applications before their deployment.

B. Contributions

In this paper, we present a tool, Taskify, which supports implementation, testing, and debugging of task-based intermittent programs on a general-purpose computer. Taskify provides the following features: (i) implementing computation-based intermittent programs using a domain-specific language (DSL), (ii) debugging the programs without deploying them into a target platform, and (iii) automatically transforming the programs written in the DSL into C programs linked to a task-based run-time environment. As part of Taskify, we provide a DSL, i.e., TaskDSL that supports a general task-based programming model while hiding the details of the existing task-based run-times. With the debugger for TaskDSL, Taskify enables engineers to catch and eliminate bugs and wrong control-flow declarations in the programs written in TaskDSL. The programs in TaskDSL are automatically transformed into C programs that can be linked to InK [19], i.e., a de facto task-based run-time environment. The generated C programs can be deployed into a target hardware platform to be executed despite frequent power failures. Taskify is implemented as an Eclipse plugin. It has been evaluated on three intermittent applications.

This paper is structured as follows. Section II provides the background on which Taskify is built. In Section III, we describe our tool, while Section VII gives the implementation details of the tool. Sections IV, V and VI provide the details of its core technical parts. Section VIII reports on our evaluation with case studies. Section IX presents a reflection on our tool. In Section X, we conclude the paper.

II. Background & Related Work

A newly emerged class of embedded sensors can operate solely on ambient [1], [2], [21] and/or dedicated wireless energy [22], [23]. The main components of such sensors are: (i) an *energy harvester* (converting ambient energy into electric current), (ii) *low-power computing module* with on-board non-volatile data storage, and (iii) *low power communication module*—ideally by means of (ambient) backscatter [24]. Typical examples of such sensors are Wireless Identification and Sensing Platform (WISP) [25], which is powered wirelessly by the distant RFID reader, and Flicker [26] that can be powered using several configurations from solar to piezoelectric energy harvesting. These platforms comprise an FRAM-based ultra-low-power micro-controller from Texas Instruments, e.g., MSP430FR5969 [27], that uses a combination of volatile (e.g., SRAM) and non-volatile (e.g., FRAM) memory. FRAM memory can be used to store information that persists upon power failures.

A. The Progress of Computation and Memory Consistency

The energy availability from (ambient) energy harvesting is unpredictable and minuscule [28]. An example trace of energy availability from RFID reader to WISP tag (at a 2 m distance) is given in [8]—only ≈100 ms in a period of 250 s the WISP was actively powering the microcontroller from the RFID reader. This shows that the frequent loss of the computation state is an inevitable phenomenon, which is also justified by the distribution of computation cycles available for WISP during regular energy harvesting operation [29]. Therefore, to be able to progress the computation from where it leaves after each power failure, engineers need to design systems that cope with such failures.

The restart of a computation block after a power interrupt can be catastrophic and exhibit side-effects [11] (denoted as non-volatile memory inconsistency). On a high level, when sensor needs to execute an operation that has *Write-After-Read* dependency on a variable stored in non-volatile memory (for instance {`x++; vector[x]=v;`}) and if the power failure occurs after `x++`, the computation block might be restarted that leads value `x` to be increased twice, introducing an inconsistency.

B. Intermittent Computing Approaches

There are two main approaches that ensure the progress of computation and memory consistency of intermittent programs. In checkpointing approaches, a program is instrumented, either by an engineer or a compiler, with instructions

```
1  // task-shared variables.
2  __shared(int data[10]; int i;);
3  // the entry task
4  ENTRY(Start){
5   // sample sensor
6   int read = sample();
7   // data[i] = read
8   __SET(data[__GET(i)],read);
9   ...
10  NEXT(Last); // next task is Last
11 }
12
13 TASK(Last){
14  ...
15  NEXT(null); //task finishes
16 }
```

Fig. 2. Pseudo code of an InK application.

to save (or checkpoint) the program state in non-volatile memory [8], [15]–[17]. In general, the program state contains the general and special-purpose registers of the micro-controller and the content of the volatile memory, i.e., the program stack. Therefore, checkpointing causes considerable overhead and prolongs program execution time. It can be performed dynamically at run-time by monitoring the supply voltage and checkpointing the program if the measured voltage level is under a predefined threshold. Voltage monitoring is however costly in terms of energy expenditure (naturally, measuring energy has a non-negligible energy cost). If other than conventional ADC measurement circuits are needed, the hardware cost is also non-negligible. A task-based programming model that requires engineers to structure their program into idempotent tasks and that provides access to the non-volatile memory through input-output channel abstractions between tasks is first proposed by Colin et al. [12]. Following this study, several task-based run-times are proposed [18]–[20], [30]. These studies showed that, from an energy perspective, the task-based programming model is a structured and more efficient way than checkpointing. Among the proposed run-times, we consider the most recent task-based run-time, InK [19].

C. InK Task-Based Run-time Environment

InK requires the engineer to divide the computation into tasks and define the task-shared variables and the control flow. An InK application is a C file that is developed using the C macro definitions provided by the InK library. Fig. 2 shows the code of a sample InK application. In InK, `__shared` keyword is used to declare persistent variables shared among tasks (Line 2). `ENTRY` defines the first task in the computation (Lines 4-11). Successive tasks are implemented using `TASK` blocks (Lines 13-16). `NEXT` is used to switch control to the corresponding task (Lines 10 and 15). `__GET` and `__SET` are used to manipulate task-shared variables (Line 8). InK application is compiled by the C compiler and linked with the static InK library to be deployed into the target hardware.

InK employs static versioning via *double buffering*. In InK, task-shared variables are allocated in non-volatile memory by creating two versions of each variable. When a task is being executed, InK run-time also creates a scratch copy (a third version) of each shared variable the task modifies, so that the original copies in non-volatile memory remain unmodified. Before task execution, the original versions of the variables are loaded into those scratch copies. During task execution, only scratch copies are modified. Upon power failure, the task can safely be restarted since the original values of the shared variables remain unchanged. Upon successful completion of the task, the scratch copies in volatile memory are first copied into the temporary buffer in non-volatile memory (commit phase 1). Then, the values in the temporary buffer are copied into the original buffer so that the values of the task-shared variables are updated (commit phase 2). Commit phase 2 can be restarted upon a power failure without causing any memory inconsistency since there is no write-after-read dependency during the copy operation. After commit phased 2 is finished, the next task in the control flow is executed. The aforementioned operations ensure the atomic completion of the tasks and the consistency of the task-shared variables.

D. Tools, Debuggers and Testbeds for Intermittent Systems

There are tools, debuggers, and testbeds proposed for batteryless and intermittent systems in the literature. For instance, Hester et al. [31] proposed hardware recording the energy patterns of energy harvesting environments. The hardware also replays the recorded energy pattern to observe the behavior of the batteryless applications on target hardware. Colin et al. [32] proposed a hardware/software tool that supports debugging applications during their intermittent execution on a target platform without interfering with their energy level. Surbatovich et al. [33] proposed a program analysis tool for detecting I/O-related bugs at compile time. Geissdoerfer et al. [34] proposed a testbed of several batteryless devices that can record and replay the energy characteristics of environments. To the best of our knowledge, Taskify is the first tool that supports catching bugs related to task-splitting and functional correctness of computation-based applications before target deployment.

III. Taskify: Tool Overview

The process in Fig. 3 presents an overview of our tool. In Step 1, *Write the Task-based Program using TaskDSL*, the engineer writes a task-based intermittent program on a general-purpose computer. To do so, Taskify provides TaskDSL, with a custom Eclipse editor, which hides the details of the existing task-based run-time environments. The output of Step 1 is a TaskDSL specification.

Once the engineer writes the task-based intermittent program, in Step 2 (*Debug the Task-based Program written in TaskDSL*), Taskify supports debugging of the program to catch and eliminate bugs and wrong control-flow declarations in the program. The output of Step 2 is a bug-free TaskDSL specification.

In Step 3, *Generate C Program from TaskDSL Program*, Taskify automatically transforms the program written in

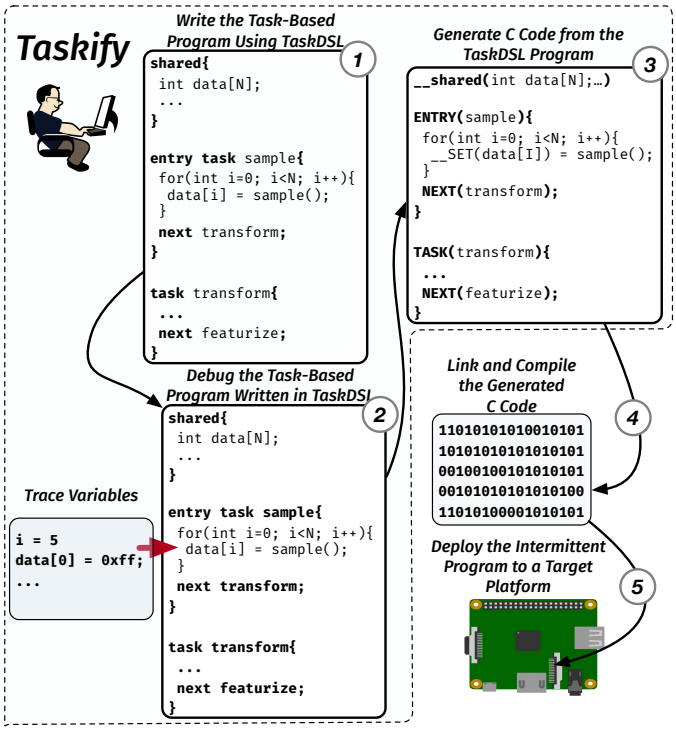

Fig. 3. Taskify Tool Overview. Engineers can develop and debug the task-based intermittent software on a general purpose computer. The developed task-based application is automatically transformed into an InK program that is compatible to be linked with the InK library.

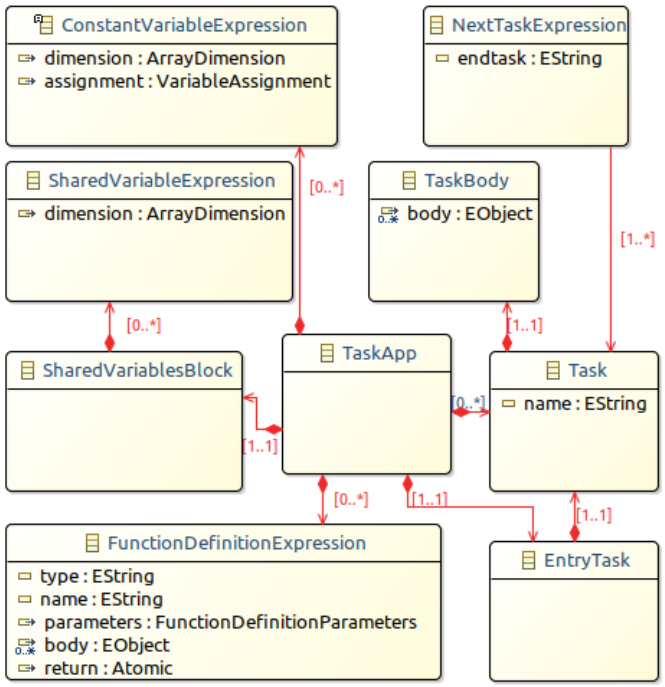

Fig. 4. Part of the TaskDSL Metamodel.

TaskDSL into C program which can be linked to an underlying run-time. Steps 1, 2 and 3 are directly supported by Taskify.

In Step 4, *Link and Compile the Generated C Code*, the engineer first links the generated code to InK [19] and then compiles it. In Step 5, *Deploy the Intermittent Program to a Target Platform*, the engineer deploys the compiled C program into a target device. In the rest of the paper, we elaborate each step in Fig. 3 using an example intermittent program written in TaskDSL.

IV. Specification of Task-based Intermittent Programs

As the first step, the engineer writes the task-based intermittent program using TaskDSL supported by our custom Eclipse editor in Taskify. TaskDSL captures the basic programming constructs in order to represent a task-based programming model; it is aligned with the InK run-time environment [19]. These programming constructs enable the declaration of (i) code blocks that are executed atomically, (ii) variables that are shared by the atomic code blocks, and (iii) control-flow that determines the progress of the computation.

Fig. 4 shows part of the TaskDSL metamodel. In this metamodel, `TaskApp` represents an intermittent program that executes multiple tasks. `Task` represents an arbitrary computation that has all-or-nothing semantics. This computation can be interrupted by arbitrarily-timed power failures. Upon power failure, the task is restarted by rolling back all the modifications it has done in the memory. The first task to be executed in the intermittent program is an `EntryTask`, while `NextTaskExpression` refers to the next task to be executed in the intermittent program. `Task` includes `TaskBody` which contains the computation of a task. The computation of a task manipulates shared variables, which are represented by `SharedVariableExpression`. Shared variables can be seen as a communication mechanism among tasks. All shared variables are declared under `SharedVariablesBlock`. Underlying run-time environments allocate non-volatile memory to the variables in this block. These variables are also versioned by employing a double-buffering scheme [12], [19], so that tasks manipulate scratch variables (see Section II-C) instead of their original versions. Upon successful completion of the task, the values of scratch variables are atomically committed to the original versions of these variables. If the task execution is interrupted by a power failure, the original variables remain unchanged—the memory, and, in turn, the task-shared variables are always kept in a consistent state. `ConstantVariableExpression` represents a constant variable in the program. Apart from task definitions, an intermittent program can have functions (`FunctionDefinitionExpression`) which are called by tasks. These functions are not allowed to read and write task-shared variables; they can only process their parameters and return values.

A sample program in TaskDSL is presented in Fig. 5. The metamodel in Fig. 4 is reflected via keywords `task`, `entry`, `SHARED`, `CONSTANT`, `next` and `end` in Fig. 5. Tasks are implemented within `task` code blocks (Lines 20-28 and 30-35 in Fig. 5). Tasks communicate with each other by

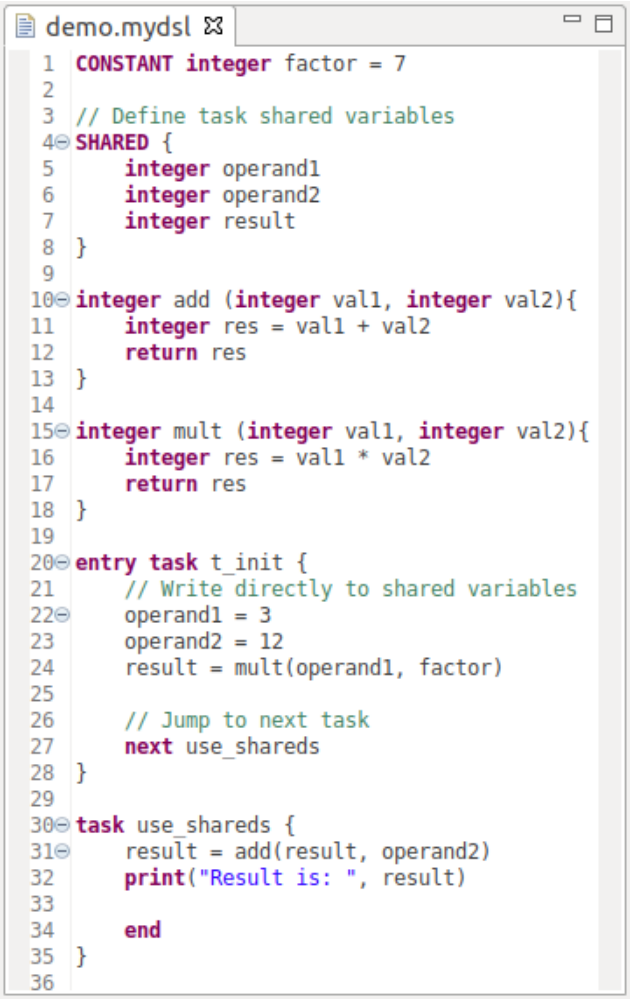

Fig. 5. A sample TaskDSL specification.

manipulating task-shared variables declared within `SHARED` block (Lines 4-8). Currently, TaskDSL supports string, integer, float and boolean data types and array definitions. `CONSTANT` keyword is used to define constant (read-only) variables with a global scope (Line 1). The example program has two tasks, i.e., `t_init` and `use_shareds`. The first task to be executed is specified by `entry` keyword (Line 20). This task initializes task-shared variables `operand1` and `operand2` to be multiplied. The result is stored in task-shared variable `result` (Line 24). The second task adds task-shared variables `operand2` and `result` and stores the result in `result` (Lines 30-35). TaskDSL supports reusable functions that can be called from tasks. These functions cannot manipulate task-shared variables; they can only return values to tasks. Multiplication and addition operations in tasks `t_init` and `use_shareds` are performed with functions `add` and `mult` (Lines 10-13 and 15-18).

The control flow is specified via keywords `next` and `end` (Lines 27 and 34). `next` is used to finalize the corresponding task and switch to the next task. `end` is used to finish computation and return control to the run-time environment.

V. Debugging of Intermittent Programs in TaskDSL

The engineer debugs the intermittent program in TaskDSL to catch and eliminate bugs, e.g., mistakes in the control flow (Step 2 in Fig. 3). Taskify debugger enables the execution and tracing of variables of TaskDSL programs on a general-purpose computer before the target hardware deployment. To this end, Taskify debugger implements execution procedures for all the elements of TaskDSL metamodel partially represented in Fig. 4. Taskify debugger generates an abstract syntax tree (AST) of the program written in TaskDSL. The generated AST is traversed to call the corresponding procedures of the nodes in the AST.

Fig. 6 shows the debugging environment in Taskify. The top-left window lists variables and their current values during debugging, while the middle-left window highlights breakpoints in the TaskDSL specification. The bottom-left window in Fig. 6 shows the current task in the control flow.

VI. Generation of C Code using InK Library from TaskDSL Specifications

TaskDSL specifications are automatically transformed into C code using the InK run-time library (Step 3 in Fig. 3). For each element of the TaskDSL metamodel in Fig. 4, we implemented the necessary routines that generate the corresponding C code. Taskify traverses AST of the intermittent program in TaskDSL, whose nodes are elements in the TaskDSL metamodel. During the traversal, the code generation routines for each AST node is executed.

Fig. 7 shows the C code generated from the example TaskDSL specification in Fig. 5. The generated code includes the InK library (Line 1 in Fig. 7). The generation of C code from TaskDSL is quite straightforward for some of the metamodel elements in Fig 4 such as `SharedVariablesBlock` and `SharedVariableExpression` (Lines 4-8). For each task in the TaskDSL specification, Taskify automatically generates task declarations using the InK library (Lines 10-11). Since the InK run-time environment requires an initialization procedure, Taskify generates initialization routine `thread1_init` (Lines 13-17) in which an InK thread is created via `CREATE` and the thread is started via `SIGNAL` calls. A C function is generated for each function in the TaskDSL specification (Lines 19-27). Finally, Taskify generates the task bodies (Lines 29-34 and 36-39). The generated tasks read and write task-shared variables via interfaces `__GET` and `__SET` in InK. Taskify inserts calls to these interfaces in order to manipulate task-shared variables (Lines 30-32 and Line 37).

VII. Taskify: Implementation & Availability

Taskify has been implemented as an Eclipse plug-in. This plug-in activates the user interfaces of Taskify and provides the features *writing a task-based intermittent program*, *debugging the task-based program on a general-purpose computer*, and *generating C code from programs in TaskDSL*.

We used Xtext framework [35], i.e., an open-source software framework for developing programming languages and DSLs,

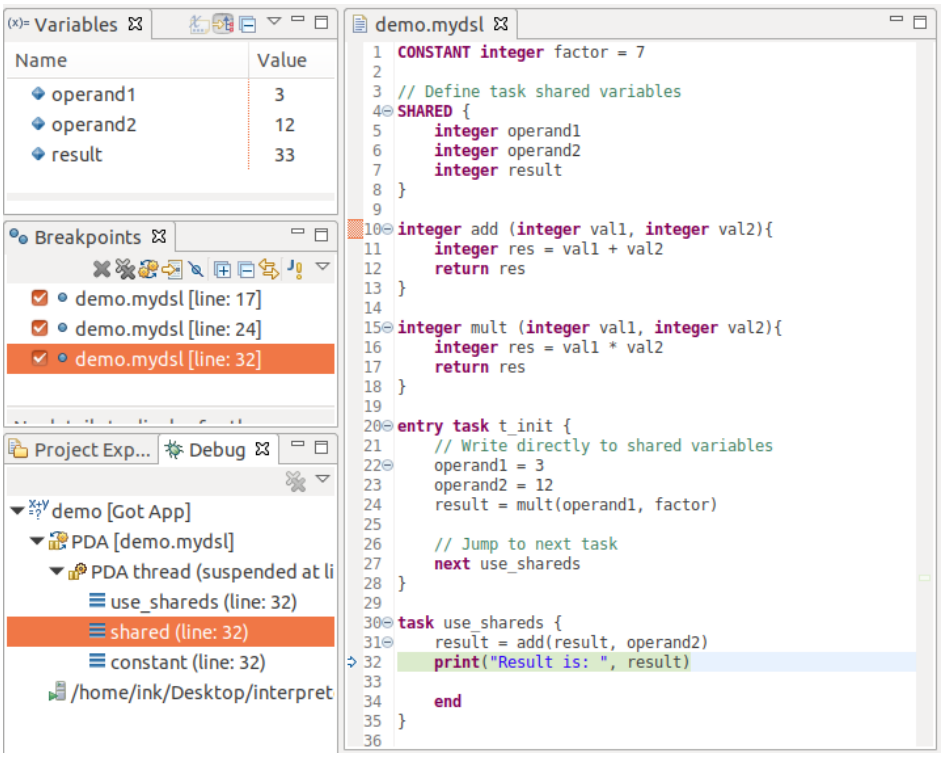

Fig. 6. A screenshot of Taskify integrated development. Taskify IDE enables the Eclipse integrated development of TaskDSL application, single-step execution of the TaskDSL program, placing breakpoints and enabling engineers to trace task-shared variables.

to develop TaskDSL. Taskify relies upon a customized Eclipse editor to write and debug task-based intermittent programs using TaskDSL.

Eclipse provides a debug framework supporting interfaces for a language-independent debug model [36]. The debug framework provides common debugging features. We used this framework to implement Taskify debugger in Java.

We used Xtend [37], i.e., a general-purpose high-level programming language for the Java Virtual Machine, to implement the automated generation of C code from intermittent programs written in TaskDSL. The Xtext project that is used to develop TaskDSL includes an empty generator stub. This stub is extended to generate C code for each entity in the TaskDSL metamodel. Xtext provides an AST of the program in TaskDSL.

Taskify is approximately 4500 lines of code, excluding comments and third-party libraries. Additional details about Taskify, including executable files and a screencast covering motivations, are available on the tool's website at:

https://github.com/tinysystems/Taskify

VIII. Evaluation

Our goal was to assess the benefits of intermittent program development with Taskify. Our evaluation aimed to answer the following research questions (RQs):

- *RQ1. Does the tool reduce the cost of the intermittent program development?*

TABLE I
THE NUMBER OF LINES AND THE SIZE OF THE .TEXT, .BSS AND .DATA SECTIONS OF THE APPLICATIONS.

Application	Type	Lines	Section size (bytes) .text	.data	.bss
Bit Count	InK Repo.	380	2707	316	4592
	Taskify	326	3616	347	5412
Cuckoo Filt.	InK Repo.	500	3106	349	4864
	Taskify	370	3748	347	5424
Cold Chain	InK Repo.	370	2628	316	11276
	Taskify	276	2830	316	18264

- *RQ2. Does the generated C code have less code size and memory requirements compared to manually developed C code?*
- *RQ3. Does the generated C code run correctly on the target platform?*

To answer the RQs above, we implemented three applications in Taskify, which are commonly used for benchmarking [12], [19]. These applications are (i) *Bitcount* counting bits in a random string, (ii) *Cuckoo Filtering* that runs a cuckoo filter over a set of pseudo-random numbers and performs the sequence recovery using the same filter, and (iii) *Cold-Chain Equipment Monitoring* emulating a temperature sensor data with a pseudo-random number, which is later compressed using the LZW algorithm [38].

```c
#include "ink.h"
#define factor 7
// Define task-shared persistent variables.
__shared(
 uint32_t operand1;
 uint32_t operand2;
 uint32_t result;
)
// Declare tasks that will be implemented
ENTRY_TASK(t_init);
TASK(use_shareds);
// Called at the very first boot
void thread1_init(){
 // create a thread with entry task
 __CREATE(15, t_init);
 __SIGNAL(15);
}
// Define helper functions
int add (uint32_t val1, uint32_t val2) {
 uint32_t res = val1 + val2;
 return res;
}

int mult (uint32_t val1, uint32_t val2) {
 uint32_t res = val1 * val2;
 return res;
}
// Implementation of all tasks
ENTRY_TASK(t_init) {
 __SET(operand1, 3);
 __SET(operand2, 12);
 __SET(result, mult(__GET(operand1), factor));
 return use_shareds;
}

TASK(use_shareds) {
__SET(result,add(__GET(result),__GET(operand2)));
 return NULL;
}
```

Fig. 7. InK Code generated from the TaskDSL Specification in Fig. 5

a) RQ1: We observed that intermittent program development in Taskify is less time-consuming and error-prone compared to manual development of C code using the InK run-time library. TaskDSL hides the details of the InK library from the engineer, while the engineer only focuses on splitting the computation into a set of tasks. He/she defines the control flow and identifies task-shared variables without dealing with the InK library declarations and interfaces. With Taskify, we eliminated most of the syntax errors while coding the three applications in TaskDSL. In addition, we eliminated all bugs related to the functional correctness before the target deployment of the applications. Compared to the three applications in the InK repository [39], our corresponding applications in TaskDSL have less number of code lines (see Table I).

b) RQ2: Code size and memory requirements of the generated codes are slightly worse than those of the applications in the InK repository (see Table I). The current implementation of TaskDSL does not support a data type for data type `char` in C. Therefore, all variables in TaskDSL occupy 4 bytes, which leads to an increased code size and memory requirements. To solve this issue, we plan to support a data type of 1 byte.

c) RQ3: To check whether the generated C code runs correctly on the target platform, we linked the generated code to the InK run-time to be deployed into TI MSP-EXPFR5969 evaluation board [40]. We observed that all three applications produce the desired outputs, which also justifies the correctness of Taskify's code generation feature.

IX. Reflections on Taskify

In this section, we discuss the limitations of Taskify and its prospective features.

a) I/O-based Applications: Taskify currently supports only the development and debugging of computation-based applications, i.e., programs without any peripheral interaction. On the other hand, most of the sensing applications are I/O driven. In order to develop and debug I/O-based applications in Taskify, I/O operations should be captured by specific keywords in TaskDSL. In addition, test scaffold code, i.e. code that mimics hardware-dependent operations, should be implemented by the engineers.

b) Other Runtimes: Taskify currently generates code only for the InK environment. We expect code generation for Alpaca [20] to be similar since Alpaca shares similar programming abstractions with Ink. In other environments such as Chain [12] and Mayfly [18], tasks communicate with each other to manipulate non-volatile memory via the so-called channel abstractions. Therefore, code generation for a channel-based runtime environment requires that channel connections among tasks be structured for each task to read the outputs of the predecessor task and to write to the inputs of the successor task—quite different than code generation for InK.

c) Checking Program Behavior: Power traces recorded from testbeds [34] can be used by Taskify to simulate and check program behavior with real power failures before target deployment. To this end, model reasoning tools, e.g., [41], can be integrated into Taskify.

X. Conclusions and Future Work

We presented a tool, Taskify, that supports the specification and debugging of task-based intermittent programs before their deployment into a target platform. The key characteristics of our tool are (1) allowing engineers to develop intermittent programs using TaskDSL, a DSL hiding the details of the existing run-time environments, (2) enabling engineers to catch and eliminate bugs and wrong control-flow declarations in the programs written in TaskDSL, and (3) automatically generating executable C code for an underlying run-time environment from the programs in TaskDSL. Taskify has been evaluated over three intermittent applications. The evaluation shows that our tool is practical and beneficial to develop task-based intermittent programs and debug them before their deployment into a target platform. We plan to (i) conduct more case studies to better evaluate the practical utility and usability of the tool, (ii) generate transformations for other run-time environments such as Chain [12], (iii) develop a testing environment that uses prerecorded power traces from real testbeds to test intermittent programs with different power

traces, and (iv) extend TaskDSL for developing I/O-based intermittent applications.

REFERENCES

[1] S. Gollakota, M. S. Reynolds, J. R. Smith, and D. J. Wetherall, "The emergence of rf-powered computing," *Computer*, vol. 47, no. 1, pp. 32–39, 2013.

[2] J. R. Smith, *Wirelessly powered sensor networks and computational RFID*. Springer Science & Business Media, 2013.

[3] R. V. Prasad, S. Devasenapathy, V. S. Rao, and J. Vazifehdan, "Reincarnation in the ambiance: Devices and networks with energy harvesting," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 195–213, 2013.

[4] M. Buettner, B. Greenstein, A. Sample, J. R. Smith, D. Wetherall *et al.*, "Revisiting smart dust with rfid sensor networks," in *Proceedings of the 7th ACM Workshop on Hot Topics in Networks (HotNets-VII)*, 2008.

[5] J. R. Smith, A. P. Sample, P. S. Powledge, S. Roy, and A. Mamishev, "A wirelessly-powered platform for sensing and computation," in *International Conference on Ubiquitous Computing*. Springer, 2006, pp. 495–506.

[6] Texas Instruments, Inc., "FRAM faqs," http://www.ti.com/lit/ml/slat151/slat151.pdf, 2014, last accessed: Jul. 28, 2017.

[7] M. Piñuela, P. D. Mitcheson, and S. Lucyszyn, "Ambient rf energy harvesting in urban and semi-urban environments," *IEEE Transactions on microwave theory and techniques*, vol. 61, no. 7, pp. 2715–2726, 2013.

[8] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 159–170.

[9] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 526–537.

[10] M. Buettner, B. Greenstein, and D. Wetherall, "Dewdrop: an energy-aware runtime for computational rfid," in *Proc. USENIX NSDI*, 2011, pp. 197–210.

[11] B. Ransford and B. Lucia, "Nonvolatile memory is a broken time machine," in *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014, pp. 1–3.

[12] A. Colin and B. Lucia, "Chain: tasks and channels for reliable intermittent programs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 514–530.

[13] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 12, pp. 1968–1980, 2016.

[14] H. Jayakumar, A. Raha, and V. Raghunathan, "Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. IEEE, 2014, pp. 330–335.

[15] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 575–585, 2015.

[16] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 17–32.

[17] V. Kortbeek, K. S. Yildirim, A. Bakar, J. Sorber, J. Hester, and P. Pawełczak, "Time-sensitive intermittent computing meets legacy software," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 85–99.

[18] J. Hester, K. Storer, and J. Sorber, "Timely execution on intermittently powered batteryless sensors," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017, pp. 1–13.

[19] K. S. Yıldırım, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester, "Ink: Reactive kernel for tiny batteryless sensors," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, 2018, pp. 41–53.

[20] K. Maeng, A. Colin, and B. Lucia, "Alpaca: intermittent execution without checkpoints," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[21] T. Soyata, L. Copeland, and W. Heinzelman, "Rf energy harvesting for embedded systems: A survey of tradeoffs and methodology," *IEEE Circuits and Systems Magazine*, vol. 16, no. 1, pp. 22–57, 2016.

[22] K. Huang and X. Zhou, "Cutting the last wires for mobile communications by microwave power transfer," *IEEE Communications Magazine*, vol. 53, no. 6, pp. 86–93, 2015.

[23] S. Bi, C. K. Ho, and R. Zhang, "Wireless powered communication: Opportunities and challenges," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 117–125, 2015.

[24] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith, "Ambient backscatter: Wireless communication out of thin air," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 39–50, 2013.

[25] M. Philipose, J. R. Smith, B. Jiang, A. Mamishev, S. Roy, and K. Sundara-Rajan, "Battery-free wireless identification and sensing," *IEEE Pervasive computing*, vol. 4, no. 1, pp. 37–45, 2005.

[26] J. Hester and J. Sorber, "Flicker: Rapid prototyping for the batteryless internet-of-things," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, 2017, pp. 1–13.

[27] Texas Instruments. (2018) MSP430FR5969 launchpad development kit. [Online]. Available: http://www.ti.com/tool/MSP-EXP430FR5969

[28] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel, "Intermittent computing: Challenges and opportunities," in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[29] J. Tan, P. Pawełczak, A. Parks, and J. R. Smith, "Wisent: Robust downstream communication and storage for computational rfids," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[30] C. Durmaz, K. S. Yildirim, and G. Kardas, "Puremem: a structured programming model for transiently powered computers," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1544–1551.

[31] J. Hester, T. Scott, and J. Sorber, "Ekho: realistic and repeatable experimentation for tiny energy-harvesting sensors," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, 2014, pp. 330–331.

[32] A. Colin, G. Harvey, B. Lucia, and A. P. Sample, "An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 577–589, 2016.

[33] M. Surbatovich, L. Jia, and B. Lucia, "I/o dependent idempotence bugs in intermittent systems," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–31, 2019.

[34] K. Geissdoerfer, M. Chwalisz, and M. Zimmerling, "Shepherd: a portable testbed for the batteryless iot," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019, pp. 83–95.

[35] Eclipse, "Eclipse xtext framework," https://www.eclipse.org/Xtext/, 2020, last accessed: April. 28, 2020.

[36] Eclipse, "Eclipse debug project," https://wiki.eclipse.org/Debug, Jan. 2020.

[37] L. Bettini, *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[38] T. A. Welch, "A technique for high-performance data compression," *Computer*, no. 6, pp. 8–19, 1984.

[39] InK, "Ink github repository," https://github.com/TUDSSL/InK/tree/master/Application, 2020, last accessed: Apr. 30, 2020.

[40] T. Instruments, "Msp430fr5969 launchpad development kit," http://www.ti.com/tool/msp-exp430fr5969, Jan. 2015.

[41] F. Erata, A. Goknil, I. Kurtev, and B. Tekinerdogan, "AlloyInEcore: embedding of first-order relational logic into meta-object facility for automated model reasoning," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 920–923.