PureMEM: A Structured Programming Model for Transiently Powered Computers

Caglar Durmaz
International Computer Institute
Ege University, Izmir, Turkey
caglar.durmaz@gmail.com

Kasım Sinan Yıldırım
Department of Computer Engineering
Ege University, Izmir, Turkey
sinan.yildirim@ege.edu.tr

Geylani Kardas
International Computer Institute
Ege University, Izmir, Turkey
geylani.kardas@ege.edu.tr

ABSTRACT

Advances in energy harvesting circuits and energy efficient architecture of processors create the potential for batteryless computing and sensing systems called transiently powered computers. These computers can only operate intermittently due to fluctuating nature of ambient energy. Intermittent operation requires a new programming model that should preserve forward progress and maintain data consistency; which are challenging. We propose a structured task-based programming model; namely PureMEM, to cope with these challenges. We discuss how PureMEM prevents interdependencies caused by the unstructured control encountered in intermittent operation, enables re-usability of the tasks, provides dynamic memory management and supports error handling. We also present intermittent programs to exemplify the features of PureMEM.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; • **Software and its engineering → Runtime environments**;

KEYWORDS

Task-Based Programming Model, Transiently Powered Computers, Embedded Systems and Software, Structured Programming Model

ACM Reference Format:
Caglar Durmaz, Kasım Sinan Yıldırım, and Geylani Kardas. 2019. PureMEM: A Structured Programming Model for, Transiently Powered Computers. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3299739

1 INTRODUCTION

Advances in energy harvesting circuits and microelectronics have led to a new type of computing systems; namely transiently powered computers (TPCs), that can operate relying on ambient energy only without requiring batteries. As an example, Wireless Identification and Sensing Platform (WISP) [13] is a tiny computer that operates by using the captured energy of radio waves. The captured energy is stored in a small capacitor and if the stored energy is above a threshold, WISP wakes up to sense the environment and perform computation. When the energy in the capacitor is depleted, WISP dies due to a power failure—leading to an intermittent operation. As opposed to continuously-powered computers, TPCs follow a duty cycle composed of charge→operate→die phases [12]. Since the ambient energy is non-deterministic, TPCs experience frequent power failures which reset the volatile state of the device; e.g. stack, program counter, registers. Therefore, existing programs and libraries designed for continuously-powered computers cannot run on TPCs correctly due to the frequent loss of volatile state—power failures give rise to failed computation and incorrect results.

In order to preserve progress of computation in TPCs, the researchers proposed inserting checkpoints in the program source at compile time [1, 2, 8, 9, 12, 16], so that the whole volatile state of the processor is saved in non-volatile memory. Upon reboot, the volatile state will be recovered using the checkpointed information and the computation will be restored from where it left. However, checkpointing introduces considerable store/restore overhead due to the size of the volatile state. This issue motivated researchers to develop task-based programming models for TPCs [3, 6, 10, 17]. In these models, the program source is composed of a collection of restartable tasks, task-based control flow and input/output channels for the data flow among the tasks. Task-based programming environments do not checkpoint the whole volatile state: they execute the current task in the control flow, restart it upon recovery from a power failure, guarantee its atomic completion, and then switch to the next task in the control flow.

However, as compared to the structured programming languages such as C, existing task-based programming models provide limited abstractions and in turn expose several disadvantages. In particular, in current task-based systems (a) control flow statements lead to "spaghetti code", (b) tasks are tightly-coupled with each other since they share global input/output data that decreases their re-usability, (c) tasks do not have signatures and therefore make computation vulnerable to potential bugs and (d) automatic and dynamic memory management is not allowed that leads to a bigger memory footprint. This paper addresses these issues and introduces a structured task-based programming model for TPCs; namely PureMEM. We list the main contributions of PureMEM to the state-of-the-art as follows:

(1) Contrary to parameterless tasks and task-based control flow in existing programming models, PureMEM enables signatures for the functions and introduces function composition as a control flow structure in order to ***eliminate "spaghetti code"*** and to ***reduce bugs***. Thanks to continuation-passing style, PureMEM prevents interdependencies caused by the unstructured control and enables ***re-usability*** of the tasks.

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3299739

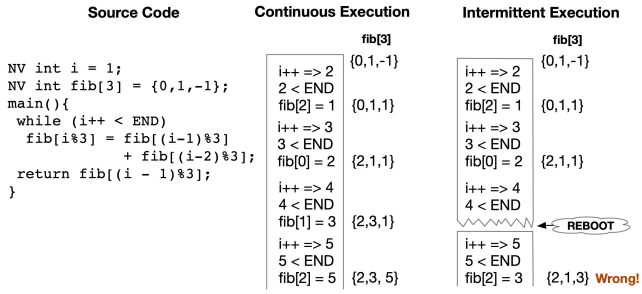


Figure 1: Intermittent execution causes errors. The program computes NTH Fibonacci number. `fib` array and `i` are stored in non-volatile memory. Intermittent execution produces the wrong result because `i` is written after read, and then system reboots before `fib` array is updated.

(2) PureMEM enables manual (dynamic) memory management that allows creation of variables which live longer by providing allocation and deallocation of nonvolatile memory. These variables can be used for data sharing among the tasks—***limiting*** the ***scope*** and ***lifetime*** of task-shared variables in contrast global scope and lifetime task-shared variables in existing models.
(3) While prior works [3, 6, 10] do not contain any constructs on ***error handling***, routines in PureMEM may return and recover errors.

The rest of the paper is organized as follows: Section 2 includes a brief discussion on TPC and existing programming models for TPC. PureMEM programming model is discussed in Section 3. Runtime environment of PureMEM is described in Section 4. Use of PureMEM is demonstrated with a case study in Section 5. Section 6 gives the related work and Section 7 concludes the paper.

2 BACKGROUND

A typical TPC starts to operate when the energy stored in its capacitor is above a predefined threshold voltage. Operation drains the capacitor quickly, TPC shuts down and reboots when the sufficient energy is stored in capacitor again. These charge/discharge cycles make the execution intermittent. Creating programs considering intermittent execution is difficult since the programmers should pay attention to ***forward progress*** of computation and ***data consistency***. First, long-running computations can not execute in a single charge/discharge cycle—to maintain the forward progress of computation, the volatile state of the device must be persisted in a non-volatile memory so that the computation can be resumed after the power is restored. However, the volatile state and the state in non-volatile memory can be different; e.g. write-after-read (W-A-R) dependencies might create data inconsistencies. For instance, the variable `i` shown in Figure 1 is written after read. When any power failure occurs before the completion of one iteration in the "while" loop, `i` might be inconsistent since it will be incremented twice.

Intermittent operation requires a new programming model that can preserve forward progress and maintain data consistency of the programs. Researchers developed two major approaches to overcome these challenges by providing checkpointing-based [1, 2, 7–9, 12, 16] and task-based [3, 6, 10] programming models. In checkpointing-based systems, programmers and/or compilers inject checkpoints into existing C codes. The injected checkpoints copy the system's all volatile state (i.e. call-stack, registers) to non-volatile memory. Therefore, checkpointing-based models have a major system overhead [3].

2.1 Task-based Programming Models

In task-based programming models [3, 6, 10], developers decompose their programs into tasks and define transitions among them. They introduce considerably less overhead because only the updated data and the following task identifier are tracked and persisted. A task is a sequence of program instructions packaged as an atomic unit of execution and a transition is an act of transferring the control from one task to another. Task-based runtimes guarantee the *atomic execution* of tasks despite power failures: (i) when the current task is finished, its modifications to the memory is committed permanently in non-volatile memory and the next task in the control flow will be executed; (ii) if a task does not complete due to a power failure, its partial modification to the memory will not be committed and the consistency of the memory will be preserved. This type of execution ensures that tasks execute atomically.

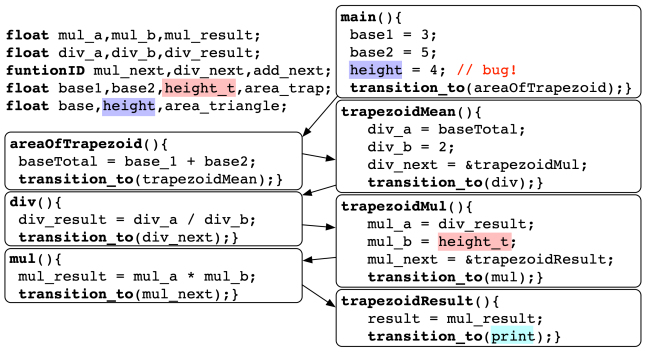
In particular, tasks read the value of W-A-R dependent variables from one memory region, and commit their modified values to another memory region. Runtime engines link the read-only and write-only memory regions with buffering, versioning and two-phase committing. The memory models of task-based programming environments hide the implementation of linking mechanisms through shared (global) variable abstractions. As an example, Alpaca [10] detects W-A-R dependent variables during compilation and provides two phase committing in run time. It copies W-A-R dependent values into a special (privatization) buffer, and then commits the modified values to main memory at the end of task execution. On the other hand, Chain [3] asks the programmer to split the variables into read-only (input) and write-only(output) variables for each task, by declaring input/output channels.

Figure 2 shows conceptual implementation of calculating the area of a trapezoid in a task-based system; e.g. Alpaca [10]. Since a task represents a transition point in code, global non-volatile variables are used for messaging among tasks. For instance, `trapezoidMean` task updates the shared variables `div_a` and `div_b` which are used for inputs of `div` before passing the control to `div` with `transition_to` statement. Result of `div` is stored in shared variable `div_result`.

2.2 Drawbacks of Existing Models

Transition methods in prior works are similar to goto statements since tasks have no arguments and the control is transfered to the following task and never transferred back (unlike subroutine calls in structured programming languages). Control flow with transitions exposes several disadvantages as listed below.

(1) *Spaghetti code:* Goto statements may lead to spaghetti code that is potentially difficult to follow and maintain [4]. If a programmer wants to change the behavior of the program or just debug the code in Figure 2, (s)he has to follow the



(a) Unstructured programming: i) Programming without task arguments is error-prone; `height_t` is not initialized before calling `areaOfTrapezoid`. ii) Since exit point of a task is defined by the programmer with goto statements, the last executed task may be maintained by following the path of goto statements. If the program is turned into a spagetti-code, maintenance will not be easy.

```
main(){
  float area=areaOfTrapezoid(3,5,4);
  print(area);
}
```

```
float div(float a, float b){
   return  = b / a;}
float mul(float a, float b){
   return  = a * b;}
```

(b) Structured Programming: i) Arguments have local lexical scope and arguments' names can be same. ii) Functions always go to one place: the statement immediately following the call (single-entry & single-exit). Last function call is in main function and can be maintained easily. The result may also be found and tested via the local variable *area*.

Figure 2: Calling tasks without arguments behaves like executing goto statements and leads to "spaghetti code".

`transition_to` statements and try to figure out the structure by considering both code and variables. However, return statements in functions of a structured programming guarantee that the flow of control is passed to statement immediately following the call of the function. Control structures (e.g., subroutines) with single-entry and single-exit make the code readable, maintainable and reusable.

(2) *Buggy code:* The limited control-flow structure of existing models forces programmers use global variables to simulate the input/output values of tasks. This may cause bugs due to the difficulty of knowing which global variables are used where. The buggy example in Figure 2 shows that a task can be called without updating its all inputs. Shared global variable `height`, instead of `height_t`, is initialized and then `areaOfTrapezoid` task receives uninitialized input.

(3) *Tightly-coupled tasks:* Preceding tasks should have the knowledge about the global variables which are used as the inputs of the following task—and/or the other way around. This kind of dependency makes the tasks tightly coupled to each other and decreases their *re-usability*. As an example, `div` in Figure 2.a is a reusable task, but the following task `trapezoidMul` is not because it depends on output of `div`.

Task-based programming models do not allow *pointers*. Since they need to version variables of W-A-R dependency at compile time to maintain data consistency. However, pointers provide dynamic access to memory—prevents versioning data at compile time. In this case, all algorithms and data structures (e.g., binary trees, linked lists) which use indirection (i.e. pointers) must be implemented without a support from the programming model. An extra effort of developing a special referencing convention is required when implementing these algorithms: declaring global arrays for each type/context and passing the indexes of elements in array as references.

Moreover, automatically managed variables like *local variables* or *manually managed variables* like heap objects are not supported in task-based programming models. All variables shared among tasks in existing models have global scope and lifetime of the entire run of the program—increasing the memory footprint of the programs.

3 PUREMEM PROGRAMMING MODEL

Decomposing the problem into small problems, solving them independently and recomposing the solutions is a common way of developing solutions for big problems. TPCs do not allow solving big problems in one charge–discharge cycle. Programmers should solve small problems in each cycle and gather the solutions piece by piece with some sort of composition. In the following subsections, we describe the PureMEM features provided for both composition and other TPC programming requirements listed in the previous section.

3.1 Composition With Routines and Closures

In order to describe how PureMEM supports composition, let us consider the composition of functions (shown in Figure 3.a) to compute the area of trapezoids in C language. While the functions `areaOfTrapezoid` and `mean` orchestrate the routine flows and pass the initial data, the other functions `add`, `div` and `mul`, perform the computations. Figure 3.b shows the similar flow of control and computation by closure sets in PureMEM routines. In this pattern, `mul` and `div` routines can also be re-used for computing the area of a triangle by composing them with an other form. **Code re-use**, which is a main concern of many developers, is also implemented by routine composition in PureMEM.

Routine as a control structure follows the common quoted rule of structured programming that each control structure should have only one entry point and one exit point. The control is always transfered to the routine on top of the closure stack, which is a data structure provided by PureMEM runtime to stack the control transfers (continuations) determined by routines. Since every routine gets a flow datum and returns a flow datum, they can be composed regarding to the arity of partially applied routines through closures. Type checking is not needed when composing closures since PureMEM has only one data type, Result (see the details in Section 3.2). Pipe function in Figure 3.a performs left-to-right function composition where as C composition is performed right-to-left. Pipe is often easier to read in evaluation order. For instance, `mean` in Figure 3.b pipes (pushes) two closures to the closure stack with the `b2` flow data. PureMEM runtime engine calls the routine `add` with the values of `b1` and `b2`, and then the routine `div` executes with the value of `float_2` and the result value of routine `add`.

Routines in PureMEM must have at least one input and one output. Last argument is called the **flow argument** and others are called as **configuration arguments**, if any. While the flow argument is supported by the preceding routine, the configuration arguments are set by parent routine via a closure. In other words, PureMEM closure defines the environment of a routine when it was created where flow data is the input data which triggers the

Figure 3: Control flow with compositions **(a)** Function composition in programming language with a direct-style: C language **(b)** PureMEM program computes area of trapezoid via closure compositions. Closures are records of partially applied routines. For instance, `div` is partially applied with a divisor `float_2` in routine `mean` by a helper function `cc_div` which is created by `ROUTINE(div,2)` signature definition. Every closure has a free variable called flow variable which is symbolized as '_ ' in creation of closures. **(c.i)** is the state of Closure Stack data structure when PureMEM runtime model with the continuation-passing style executes the routine `add`. **(c.iii), (c.ii)** are the conceptual states of call-stacks when a C based environment executes `add` function with and without tail-call optimization respectively.

execution of the routines. Function compositions in a direct-style programming language, e.g. C, are naturally realized as tail-calls; so are all calls in Figure 3.a. Tail calls can be implemented without adding a new stack frame to the call stack like Figure 3.c.iii, instead of standard call-stack in Figure 3.c.ii, because there is no statement left to execute in functions `mean` and `areaOfTrapezoid` while returning from `add` function. Optimized tail-calls may be considered as goto statements with parameters [14]. Although the cost of function calls are considered to be significant, this style makes the procedure calls as efficiently as goto statements, therefore enabling efficient structured programming.

Similarly, PureMEM implementations always provide tail-call optimization because continuation-passing (composition) with closures is a tail-call. It passes parameters to the following routines and the control flow is not returned to caller. Figure 3.c.i shows that closure stack while routine `add` executes has even less overhead in memory than the call-stack in Figure 3.c.iii.

```
INTERMITTENT_TYPE (Surface,2);
INTERMITTENT_FIELD(Surface,0,float,area);
INTERMITTENT_FIELD(Surface,1,char,color);
GLOBAL(areaOfSurface, sizeof_Surface());
ROUTINE(saveSurface, 3);//3 is arity of routine
```

```
Float saveSurface(Float b1, Float b2, Float h){
 GET_Surface_area(Float area, areaOfSurface);
 return pipe(h,
        cc_areaOfTrapezoid(b1,b2,_),
        cc_assignFloat(area,_));
}
```

```
Float assignFloat
      (Result l, Result r){
 GET_Float_val(float v_r, r);
 SET_Float_val(l,v_r);
 return r;
}
```

Figure 4: Compound data representation and the use of global variables.

3.2 Data Type, Variables and References

PureMEM programmers write and read the output values of routines in memory locations which are protected against power failures by the PureMEM memory system (see the details in Section 4.2). Memory locations can be allocated with a size value dynamically or statically with `nv_alloc` (see Figure 3) function or `GLOBAL` (see Figure 4) keyword respectively. `SET_Float_val` and `GET_Float_val` wrapper functions are called to update and read the values stored in corresponding memory locations specified with the variables `a` and `b` in `mul`, `div`, `add` in Figure 3. By providing allocation methods and a referencing mechanism, PureMEM memory system introduces an extra level of indirection to non-volatile memory to ensure safe memory access despite intermittent execution break data consistency.

3.2.1 PureMEM Data Type: PureMEM programming model has one data type; namely *Result*, whose value is an integer. All type definitions trough `INTERMITTENT_TYPE` (see the usage of it in Figure 3) must be considered as aliases of Result type, similar to `typedef` keyword in C language. PureMEM does not provide type checking functionality but `INTERMITTENT_TYPE` definitions can be used for refactoring and benefiting from helper functions; e.g. `sizeof_`, `GET_`, `SET_`. It is worth mentioning that current type system is fairly simple; however higher-level abstractions on type system; e.g. type checking, can also be incorporated—which is out of scope of this paper.

3.2.2 Values of References: The integer values of Result variables are grouped into three categories; Box, Success and Failure. *(i) Box values* represent any valid storage chunk index in PureMEM Memory Model (see Section 4.2). Box values are references enabling a program to indirectly access a particular datum in PureMEM memory model. *(ii) Success value* is a constant integer that is bigger than the maximum chunk index which is used for passing the control to the following routine without a datum or error. *(iii) Failure values* are integers bigger than Success constant. They represent a particular error which can be passed to the following routine and handled in routines. PureMEM runtime engine may return failure values like OUT_OF_MEMORY to the routines whereas developers may also define error values in PureMEM and pass them in continuations (see Section 3.3).

3.2.3 Compound Data Representation: Compound data representation is just like any other `INTERMITTENT_TYPE` definitions but with multiple `INTERMITTENT_FIELD` definitions; `Surface` in Figure 4 has two fields `area` and `color`. All fields can be accessed by `GET_` and `SET_` prefixed functions created by PureMEM.

```
ERROR(ERR_DIVISION_BY_ZERO);
Float BayesTheorem(Float p_b_a, Float p_a, Float p_b){
 return tryPipe(p_b, cc_mul(p_b_a,_),cc_div(p_b,_),
                cc_finalRoutine(_));
}
Float bind(Closure c, Any a){
  if(isError(a)) return a;
  return pipeClosure(c, a);
}
Float div(Float a, Float b){
  float a_val = GET_Float_val(a);
  if (val_a == 0) return ERR_DIVISION_BY_ZERO;
  float b_val = GET_Float_val(b);
  Float result = new_Float(b/a);
  return result;
}
Float mul(Float a, Float b)…
Any finalRoutine(Any a){
  if(isError(a))
    return pipe(a,printScreen(_),
                  newInputsFromKeyboard(_));
  return a;
}
```

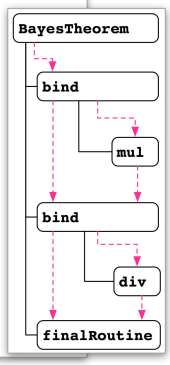


Figure 5: Error Handling: Code defines the division by zero error with the keyword ERROR and returns (similar to throw) in div routine. System function tryPipe binds the routines mul and div with system routine bind. Routine bind checks the outputs of them. If an error is returned, binded routines are all bypassed and the finalRoutine prints the error message and waits for a new input. If no error is returned, finalRoutine just passes the valid result.

3.2.4 Routine Signatures: A PureMEM routine is a C function which returns a value and has at least one argument. All arguments and the returned value must be of type Result. Because there is only one type in PureMEM, only the arity of routine with ROUTINE keyword is enough to define the signature. However, it is a good practice to create and use INTERMITTENT_TYPE definitions (aliases) in function signatures for readability and maintainability. ROUTINE definitions lead to construction of helper functions prefixed as cc_ to create closures easily in pipe functions (see some usages in Figures [3.b, 4 and 9]).

3.3 Error handling

PureMEM routines may not only detect but also recover from the system errors and user defined errors with the help of PureMEM standard library. Since PureMEM routine is packaged as a unit, error handling like try-catch mechanism can be built over routines. Figure 5 shows the usage of bind routine of tryPipe function. The system routine bind checks the flow argument whether it holds an error or a valid value. It creates a closure in closure-stack with the configuration argument when the flow data is valid. Otherwise, it passes the error data to the next routine. Error value passes through all bind routines and reach the finalRoutine. Since tryPipe abstraction provides error checking before calling the routines, error checking is discarded in mul and div routines. All possible errors in the tryPipe block are checked and handled only in finalRoutine.

4 PUREMEM RUNTIME ENVIRONMENT

4.1 Forward Progress and Memory Consistency

PureMEM system can be in one of five states shown in Figure 6. One routine in a PureMEM program is marked as the initial routine and executed when the device is powered for the first time in Initial

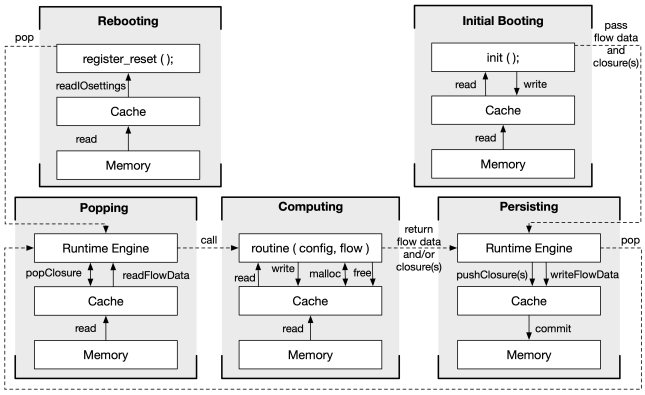


Figure 6: States of PureMEM runtime environment Dotted arrows shows the state transitions of PureMEM runtime environment. Straight arrow depicts the acts of reading and writing the memory constructs; Cache and Memory. Where Cache is stored in volatile memory, Memory is implemented on non-volatile memory of the TPC.

Booting state. After a successful initial boot, every reboot caused by a power failure starts the system from Rebooting state and executes the register reset routine. Closure compositions created by initial routine and other routines are stored in Closure Stack by Runtime Engine in Persisting state. PureMEM system runs as long as there is closure in Closure Stack and loops over Popping, Computing and Persisting states. Closure compositions returned by running routines control the flow of the program.

PureMEM runtime engine and routines do not have direct access to the non-volatile memory as depicted in Figure 6. All writing and reading operations take place via a memory caching system. In Persisting state, all updated values in Cache are committed at once as a transaction–the caching and atomic committing mechanism ensures the consistency of memory. If any power failure occurs before committing is completed, cached values will be lost and all changes on the closure stack and other memory spaces showed in Figure 7.b will not be updated at all. In this case, next reboot will cause the last closure is popped from the stack and converted into running routine with the same inputs in Computing state again. This time, changed values should be committed to keep the progress flow. Otherwise, it means that the capacitor of the system is not enough to power the states of Rebooting, Popping, Computing and Persisting for the last closure. PureMEM programmer should decompose and compose the routines in this respect.

4.2 PureMEM Memory Model

Where previous task based programming models abstract the variables in non-volatile memory, PureMEM abstracts the location of any value in the non-volatile memory to eliminate W-A-R dependencies. This low level location abstraction facilitates the implementation of other constructs in PureMEM: manual memory management, local variables, closure stack and thereby modeling in continuation-passing style.

4.2.1 Memory Location System. PureMEM implements the memory location for intermittent programming via the persistent data

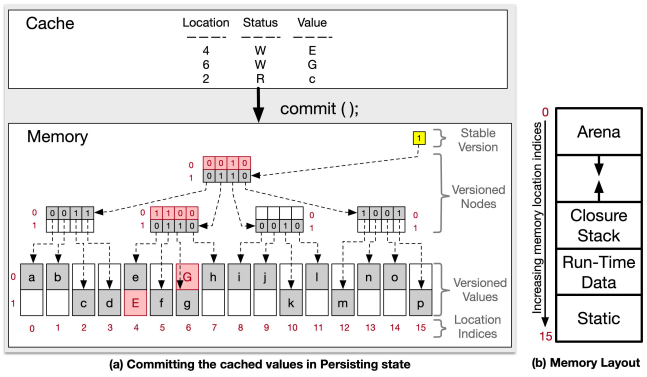


Figure 7: PureMEM memory model (a) shows the conceptual design and situation of memory model of PureMEM while updated memory locations with indexes 4 and 6 are being committed to Memory atomically via a special tree structure: Radix Balance Tree [15]. Status values in the cache `W` and `R` indicate whether the value in the position should be updated or not. **(b)** depicts the memory layout of PureMEM: Arena is the manually managed memory region. PureMEM runtime engine manages the region Closure Stack automatically. Global variables are stored in Static region.

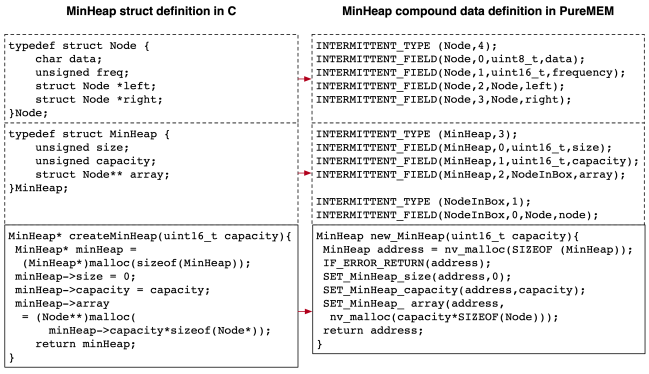


Figure 8: MinHeap and Node types in C language and corresponding types in PureMEM: It also shows how a MinHeap is created dynamically with capacity input in C and PureMEM.

structure Radix Balanced Tree (RBT)[15]. The original RBT is an immutable vector which always preserves the previous version of itself when it is modified. RBT is a tree whose nodes are fixed size arrays and leafs are the values of the vector. Every update operation on RBT creates a new RBT instance with the help of a structural sharing technique.

In PureMEM, memory is split into 4 byte chunks. The index number of the chunks are used as virtual addresses. Every node of PureMEM RBT and addressed values (chunks) in PureMEM memory have two versions; `0` and `1` (see Figure 7.a). The version of the whole memory used for Cache readings is stored in stable version bit. All write operations take place in the version 1 when the stable version is 0 or the other way around. Figure 7.a shows a RBT with a branching factor of 4. The bits in the nodes are used as pointers to the version of corresponding children's nodes. The bits on the leafs of RBT show the version of the chunk on the relevant location. The red shaded nodes and values in Figure 7.a show the updated values which will be activated after setting the stable version of PureMEM memory. Since changing the stable version of the system is the last atomic action in this transaction, the commit guarantees the consistency of the RBT and non-volatile memory. If any power failure occurs before changing the stable version, system will be rebooted with the last committed version of memory instance. This means that closure stack has not been updated during power failure, thereby last routine will be re-executed with the same input values in Arena and Static region of PureMEM memory model in Figure 7.b.

PureMEM RBT implementation enables efficient access and modification with a time complexity of O(log n). If the branching factor of RBT is chosen 16 because of having a system with 16 bit architecture, RBT can map 64MB memory in 7 levels.

4.2.2 Memory Management. PureMEM provides three types of memory allocation for power-failure immune variables.

(1) *Static memory allocation:* Global variables are allocated statically in static region of PureMEM non-volatile memory (see Figure 7.b). Their lifetime spans the execution of the program.
(2) *Automatic memory allocation:* Memory space of configuration arguments of routines are automatically allocated and deallocated by the closure stack. Because PureMEM follows continuation-passing style, memory of the configuration arguments are allocated when the continuation definition (closure) is created and reclaimed by PureMEM runtime after the routine of closure completes its execution. However, it is worth indicating that objects referenced by arguments are not automatically allocated and deallocated.
(3) *Manual memory allocation:* The objects can be created in arena dynamically via `nv_malloc` function. Unlike usual heap implementations, objects in arena are not freed individually. All objects are freed at once via freeArena function. It is similar to the region-based memory allocation models but PureMEM involves only one arena. This simple implementation runs with a small overhead of updating an integer value holds the first free location (tail) in arena. The programmers may use arena for intermediate results where they store the final results (e.g., state of the system) in static region.

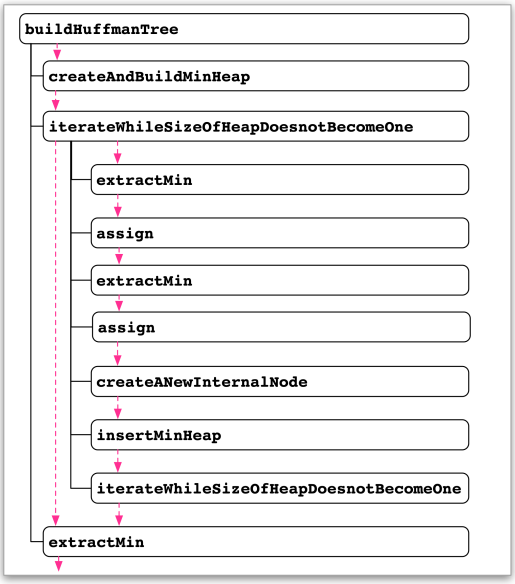
PureMEM runtime stores and updates the top of closure stack and tail of arena in the Runtime Data region. Atomicity of updating runtime data is handled by regular cache and commit mechanism. PureMEM runtime engine does not have any direct access to PureMEM memory as shown in Figure 6. All routine and runtime updates take place in cache at first, and then they are committed together as one atomic action in Persisting state.

5 CASE STUDY

A real world scenario for intermittent program development would be converting an existing C program to an intermittent program. We implemented Huffman Code to evaluate the capabilities of PureMEM programming model. Huffman Code uses minimum heap to find a prefix-free binary code with minimum expected codeword

```
MinHeapNode* buildHuffmanTree(char data[],int freq[],int size){
 // Step 1: Create local variables
   Node *left, *right, *top;
   MinHeap* minHeap;

 // Step 2: Create a min heap of capacity equal to size.
   minHeap = createAndBuildMinHeap(data, freq, size);

 // Step 3: Iterate while size of heap doesn't become 1
   while (!isSizeOne(minHeap)) {

   // Step 3.1: Extract the two items from min heap
     left = extractMin(minHeap);
     right = extractMin(minHeap);

   // Step 3.2:  Create a new internal node
     top = newNode('$', left->freq + right->freq);
     top->left = left;
     top->right = right;

   // Step 3.3:  insert the new node
     insertMinHeap(minHeap, top);
   }

 // Step 4: The remaining node is the root of Huffman tree.
   return extractMin(minHeap);
}
```

(a) BuildHuffmanTree function in C



(c) Control flow diagram of BuildHuffmanTree in PureMEM

```
ROUTINE(buildHuffmanTree, 3); //define arity of the routine
ROUTINE(iterateWhileSizeOfHeapDoesnotBecomeOne, 4);
…
Node buildHuffmanTree(Data data, Freq freq, Int size){
 // Step 1: Create arena variables
 NodeInBox left = nv_alloc(SIZE_OF(NodeInBox));
 NodeInBox right = nv_alloc(SIZE_OF(NodeInBox));
 MinHeapInBox minHeap_p = nv_alloc(SIZE_OF(MinHeapInBox));
 // Following steps 2, 3, 4
 return pipe(minHeap_p,
  cc_createAndBuildMinHeap(data, freq, size,_),
  cc_iterateWhileSizeOfHeapDoesnotBecomeOne(minHeap_p, left, right,_),
  cc_extractMin(minHeap_p,_)
 );
}
MinHeap createAndBuildMinHeap(Data d,Freq f,Int s,MinHeapInBox mhb) …
None iterateWhileSizeOfHeapDoesnotBecomeOne
 (MinHeapInBox minHeap_b, NodeInBox left, NodeInBox right, Success n)
{
 GET_MinHeapInBox_minHeap(p_minHeap, minHeap);
 if (GET_MinHeap_size(minHeap) > 1)
  return pipe(SUCCESS,
   cc_extractMin(minHeap_b,_),
   cc_assign(left,_),
   cc_extractMin(minHeap_b,_),
   cc_assign(right,_),
   cc_createNewInternalNode(left, right,_),
   cc_insertMinHeap(minHeap_b,_),
   cc_iterateWhileSizeOfHeapDoesnotBecomeOne(minHeap_b, left, right,_)
  );
 else return SUCCESS;
}
Node extractMin(MinHeap minHeap, Success n){
 NodeInBox temp = GET_MinHeap_array(minHeap);
 return pipe(n,
  cc_removeMinNodeFromHeap(minHeap,_),
  cc_returnTemp(temp,_)
 );
}
Box assign (BoxInBox l, Box r) {
 SET_BoxInBox_box(l,r); return r;
}
Node createANewInternalNode
                    (NodeInBox left_b, NodeInBox right_b, Success n){
 Node left = GET_NodeInBox_box(left_b);
 Node right = GET_NodeInBox_box(right_b);
 int l_freq = GET_Node_freq(left);
 int r_freq = GET_Node_freq(right);
 Node top = pureMEM_alloc(SIZE_OF(Node));
 SET_Node_freq(top, l_freq + r_freq);
 SET_Node_left(top, left);
 SET_Node_right(top, right);
 return top;
}
MinHeap insertMinHeap (MinHeapInBox minHeap, Node topNode) …
```

(b) BuildHuffmanTree routine in PureMEM

Figure 9: Building huffman tree in C language and PureMEM programming model: (a) shows the function in C language. (b) shows the decomposition of the function in PureMEM routines. (c) depicts the dynamically created flow diagram by PureMEM runtime regarding to the continuations returned by routines.

length. It is commonly used for lossless data compression. Minimum Heap is basically a binary tree with pointers.

The *compound data* structures used in C and equivalent structures in PureMEM for minimum heap node are shown in Figure 8. Mainly, it is a procedure of mapping the types and fields with `INTERMITTENT_TYPE` and `INTERMITTENT_FIELD`. The `new_minHeap` function in Figure 8 shows how the size of Huffman tree is configured dynamically with the help of PureMEM function **`nv_alloc`** and returning the corresponding *memory location*.

Figure 9 shows the original C code and equivalent PureMEM code of the function; `buildHuffmanTree`. Basically, the procedure is a translation from the direct style of C language to continuation-passing style of PureMEM. The flow of the C program can be decomposed into logical pieces like comments in Figure 9.a ,and then recomposed as equivalent PureMEM routine compositions in Figure 9.c. Creating continuations with pipe functions builds dynamic multi-level flowcharts like in Figure 9.b. Solving the problem in several levels increases the *code-reuse*. The routine,`extractMin` runs several times without affecting the flow because of single-entry and single-exit structure of the routines. On the contrary, prior task-based programming models cause flat flowcharts like in Figure

2.a because `transition_to` statements cannot create compositions (levels) inside of the functions dynamically.

Selection, iteration and recursion constructs of structured programming can be reduced to a PureMEM Routine which contains 'If' statement(s) and return statement with a closure sequence and a flow data. The routine, `iterateWhileSizeOfHeapDoesnotBecomeOne`, is tail-recursive function passing a continuation of itself in one of its continuations. The other selection passes only a `SUCCESS` value for running the next routine in the closure stack.

`ROUTINE` definitions of PureMEM in Figure 9.b create C functions with `cc_` prefixes which help the programmers to create closures. The `cc_` prefixed functions guarantee to call routines with all inputs at compile time and *prevent the bug* which is previously illustrated in Figure 2.a and encountered in the existing programming models.

6 RELATED WORK

Checkpointing-based and task-based computation techniques are provided by the recent efforts. i) Checkpointing: Mementos [12], QUICKRECALL [8] and Hibernus++ [1] monitor the supply voltage with hardware assistance, to checkpoint the system state. DINO [9] lets the programmer to place checkpoints independently of monitoring voltage. ii) Task-based techniques: Chain [3] provides channel-based memory model and idempotent task abstraction. Alpaca [10] uses automatic privatization and redo-logging techniques for correct intermittent programs. Recent task-based efforts [3, 10] abstract the variables in their memory model where PureMEM introduces the intermittent memory location which provides a lower abstraction than the prior studies and enables higher abstractions to create: closures, manual memory management, compound data and routine signature, error-handling and structured programming routines via efficient runtime model with continuation passing style.

Although PureMEM is not a functional programming model, it uses techniques which are used in functional programming. Tasks in task-based programming models must be atomic and idempotent [3]. Pure functions in functional programming are guarantied to be idempotent, thanks to referential transparency property [11]. Similarly, pure functions are atomic because their inputs are immutable objects [5] whose state cannot be modified after they are created (no intermediate state exists). Pure functions create new instances of the objects through some sort of copying and updating operations on persistent data structures which are implementations of immutable objects. PureMEM uses persistent data structure RBT [15] as immutable object for the representation of the whole memory space for intermittent programming. Routines cannot update system memory, they create updating records which are used for creating a new instance (reference / version) of memory by PureMEM runtime engine. Functional programming languages use closures for continuation-passing style so does PureMEM with low computation overhead.

7 CONCLUSION AND FUTURE WORK

We have shown that the low level abstractions provided by unstructured programming model for transiently powered systems bring challenges to programmers: they lead spaghetti-code which is difficult to maintain and debug, they limit the code reuse, they do not provide memory abstractions like data referencing and manual memory management. To address these limitations, we presented PureMEM: first structured task-based programming model for transiently powered systems. This paper discusses the first step of our ongoing study which tries to eliminate the disadvantages and brings the best of both checkpoint and task based programming models. Our future work will be on completing the implementation of a compiler which transforms any C code to PureMEM code automatically, after injecting tags into the original standard C code.

REFERENCES

[1] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. 2016. Hibernus++: a Self-calibrating and Adaptive System for Transiently-powered Embedded Devices. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 35, 12 (2016), 1968–1980.

[2] N. A. Bhatti and L. Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-Powered Embedded Sensing. In *2017 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. ACM/IEEE, 209–220.

[3] Alexei Colin and Brandon Lucia. 2016. 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 (OOPSLA 2016)*. ACM, Amsterdam, Netherlands, 514–530.

[4] Edsger W Dijkstra. 1968. Letters to the editor: go to statement considered harmful. *Commun. ACM* 11, 3 (1968), 147–148.

[5] Brian Goetz, Tim Peierls, Doug Lea, Joshua Bloch, Joseph Bowbeer, and David Holmes. 2006. *Java concurrency in practice*. Pearson Education.

[6] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 17.

[7] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 228–240. https://doi.org/10.1145/3079856.3080238

[8] Hrishikesh Jayakumar, Arnab Raha, Woo Suk Lee, and Vijay Raghunathan. 2015. Quickrecall: A HW/SW Approach for Computing Across Power Cycles in Transiently Powered Computers. *ACM J. Emerg. Technol. Comput. Syst.* 12, 1 (July 2015), 8:1–8:19.

[9] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, Portland, OR, USA, 575–585.

[10] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 96 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133920

[11] John C Mitchell and Krzysztof Apt. 2003. *Concepts in programming languages*. Cambridge University Press.

[12] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. 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 (ASPLOS XVI)*. ACM, Newport Beach, CA, USA, 159–170.

[13] Joshua R Smith. 2013. *Wirelessly powered sensor networks and computational RFID*. Springer Science & Business Media.

[14] Guy Lewis Steele Jr. 1977. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 annual conference*. ACM, ACM, 153–162.

[15] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. 2015. RRB Vector: A Practical General Purpose Immutable Sequence. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 342–354. https://doi.org/10.1145/2784731.2784739

[16] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16)*. ACM, Savannah, GA, USA, 17–32.

[17] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, 41–53.