Immortal Threads: Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers

Eren Yildiz[2], Lijun Chen[1], Kasim Sinan Yildirim[1]

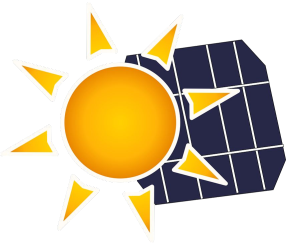
[1]University of Trento, Italy, [2]Ege University, Turkey

16th USENIX Symposium on Operating Systems Design and Implementation
July 11–13, 2022, Carlsbad, CA, USA

Energy Harvesting Batteryless Devices

- Future sensing devices are **tiny**, **sustainable** and **run forever**!

Radio Frequency

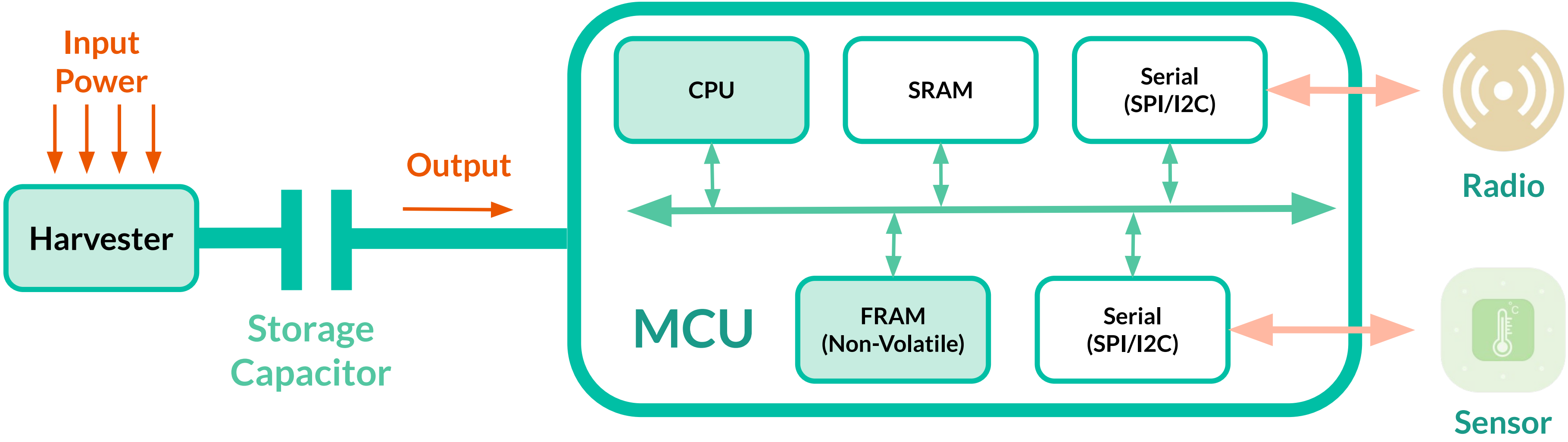


Solar

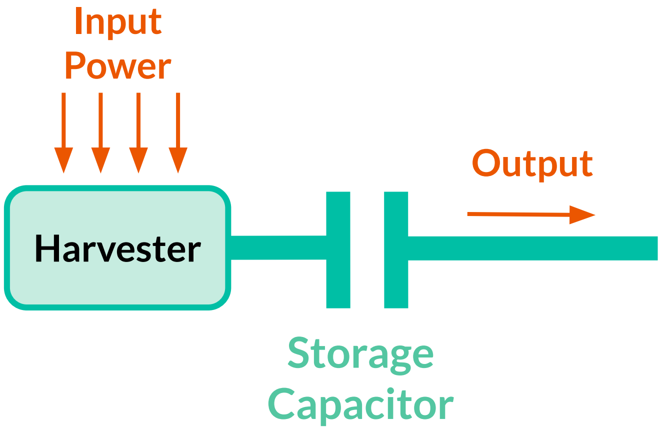
Camaroptera
[ACM TECS’22]

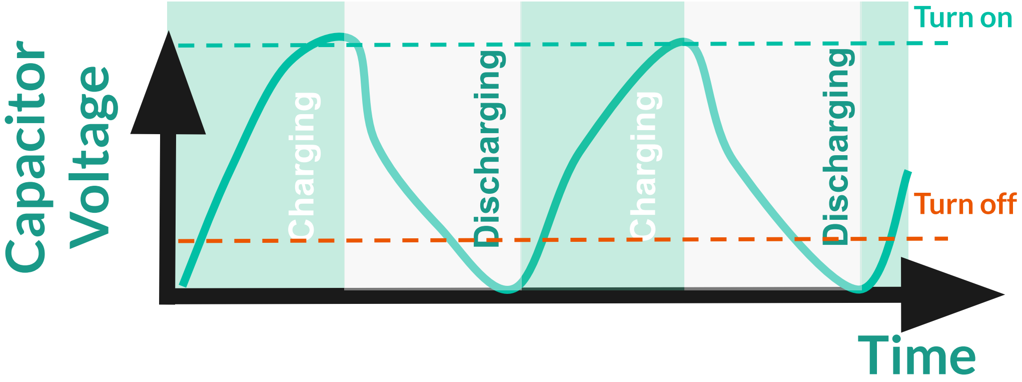
Flicker
[ACM SenSys’17]

A Typical Batteryless Sensor Architecture



A Typical Batteryless Sensor Architecture





Power Failures - Intermittent Execution

```
int i=0;
char buf[10];
main() {
 while(1)
  for(i=0..9){
   buf[i++]=read();
  }
}
```

```
main()
 while(1)
  for(i=0..9)
```

Failure

```
main()
 while(1)
  for(i=0..9)
   buf[i++]=read();
```

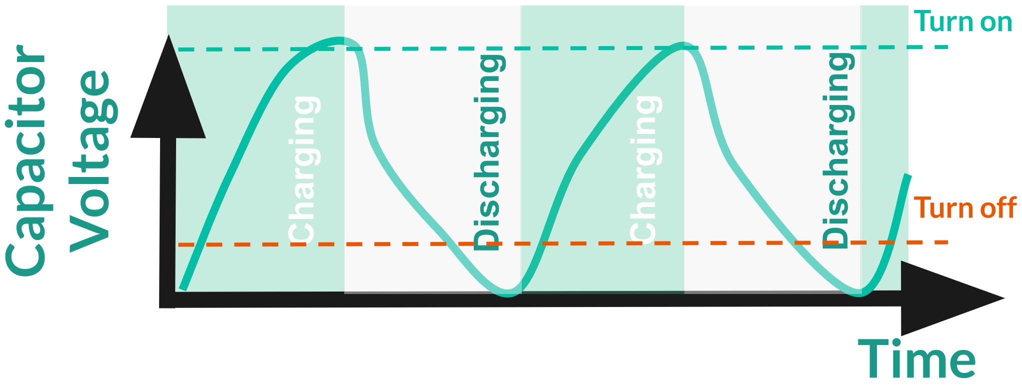
Failure

```
main()
 while(1)
  for(i=0..9)
```

Failure

```
...
```

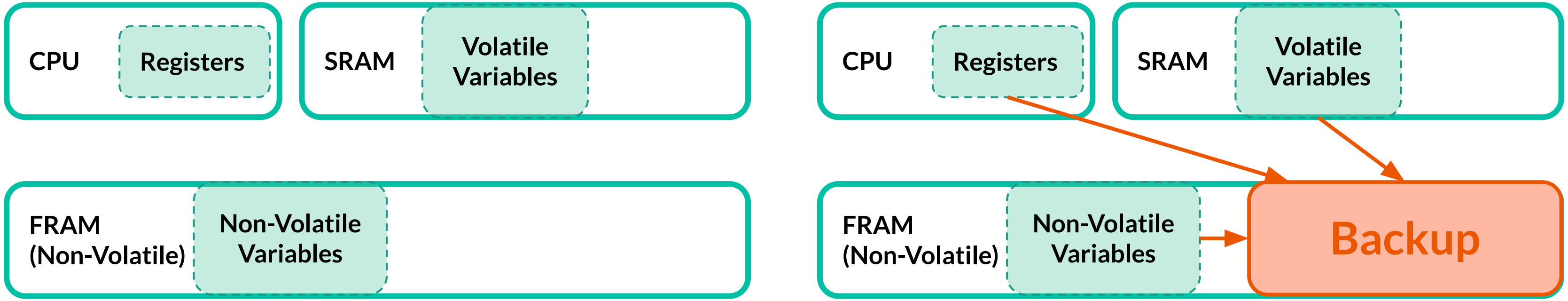
No **forward progress**/memory consistency



Outline

- Introduction
- Prior studies & Problems
- Problem Statement
- ImmortalThreads
    - Overview
    - Implementation
- Evaluation
- Conclusion

Program State - Backup and Recovery



To ensure **forward progress/memory consistency**

Checkpoints vs Tasks

```
void conv(){
 int a[N]; int b[K];
 int out[NK+1];

 for (i=0;i<NK+1;i++){
  for (j=0;i<K;j++){
   out[i]+=a[i+j]*b[K-j-1];
   checkpoint();
  }
 }
}
```

Easy/more backup overhead

```
Task init{
 write(i,0);
 next(t0);
}
```

```
Task t0{
 if(i<NK+1)
  next(t1);
 else
  next(init);
}
```

```
Task t1{
 if (j<K)
  next(conv);
 else{
  write(i,i+1);
  write(j,0);
  next(t0);
 }
}
```

```
Task conv{
 write(out[i],out[i]+a[i+j]*b[K-j-1]);
 write(j,j+1);
 next(t1);
}
```

Programmer burden/more efficient

Checkpoints vs Tasks

```
Task init{
 write(i,0);
 next(t0);
}
```

```
Task t1{
 if (j<K)
  next(conv);
 else{
```

```
Task t0{
```

Significant problems in developing event-driven applications

```
    checkpoint();
   }
  }
}
```

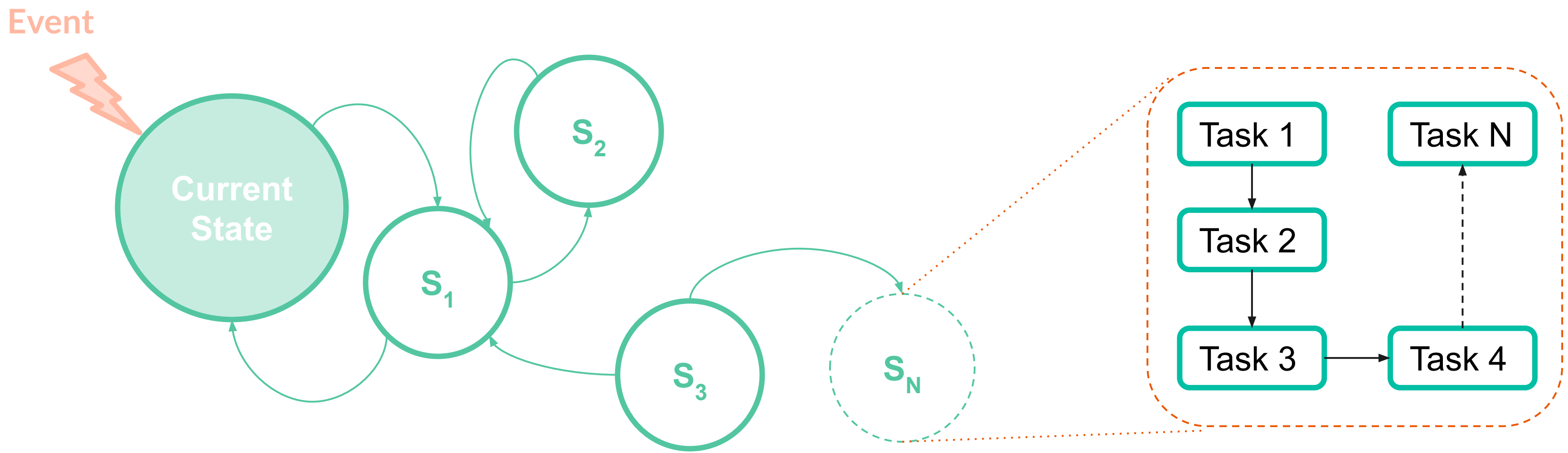
```
 write(out[i],out[i]+a[i+j]*b[K-j-1]);
 write(j,j+1);
 next(t1);
}
```

Easy/more backup overhead

Programmer burden/more efficient

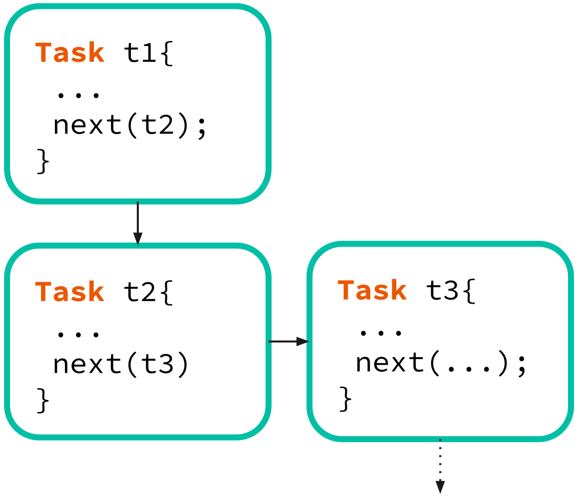
Event Handling Complexity

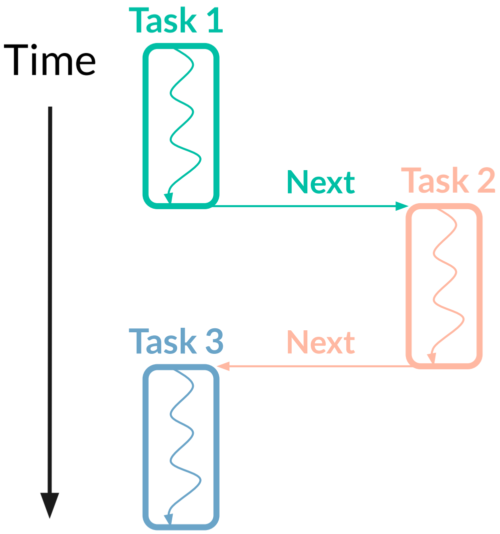
State and **transitions** management ➕ **Task partitioning** and **control flow**



Limited Concurrency

Tasks are **atomic** by definition: non-preemptive and stackless concurrency





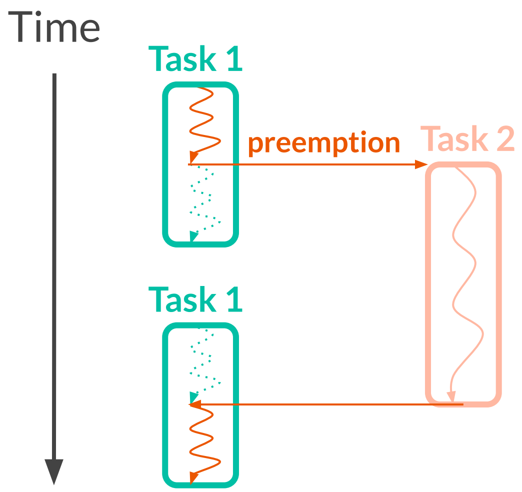
Limited Concurrency

Stackful concurrency

- Programming expressiveness
  - Blocking on events
  - Trigger new threads of execution
  - Notify the completion of event processing.

```
Task t1{
 ...
 some computation
 ...
}
```

```
Task t2{
 ...
 wait event
 process event
 signal completion
 ...
}
```



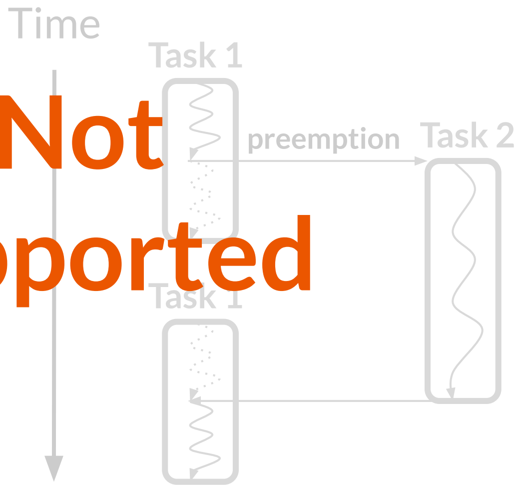
Limited Concurrency

Stackful concurrency

- Programming expressiveness
  - Blocking on events
  - Trigger new threads of execution
  - Notify the completion of event processing.

```
Task t1{
  ...
  some computation
  ...
}
```

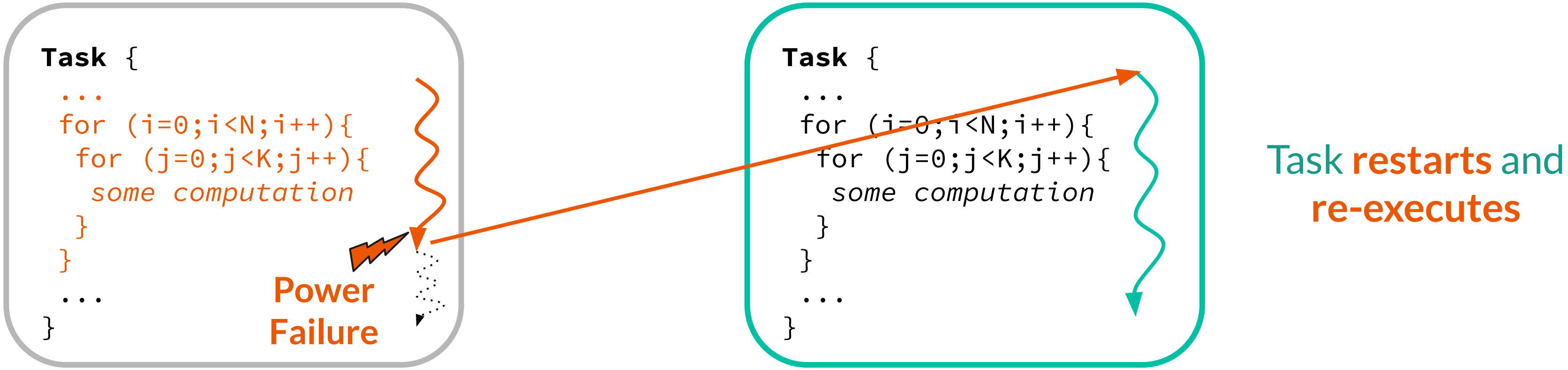
```
Task t2{
  ...
  wait event
  process event
  signal completion
  ...
}
```



Not Supported

Wasted Progress and Energy

Partial execution of tasks (due to power failures) leads to loss of computational progress.



Outline

- Introduction
- Prior studies & Problems
- Problem Statement
- ImmortalThreads
  - Overview
  - Implementation
- Evaluation
- Conclusion

Problem Statement

We need a *programming model* that

- Has **no cognitive load** and **lightweight** as task-based model

- Has flexibility of the **stackful concurrency** (**preemption + multithreading**)

- Has **minimal wasted progress** upon a power failure

Problem Statement - Immortal Threads

We need a *programming model* that

- Has **no cognitive load** and **lightweight** as task-based model

- Has flexibility of the **stackful concurrency** (**preemption + multithreading**)

- Has **minimal wasted progress** upon a power failure

Pseudo-stackful Preemptive Multithreading

Outline

- Introduction
- Prior studies & Problems
- Problem Statement
- ImmortalThreads
  - Overview
  - Implementation
- Evaluation
- Conclusion

Immortal Threads

Programmers develop programs in a **multithreaded fashion**

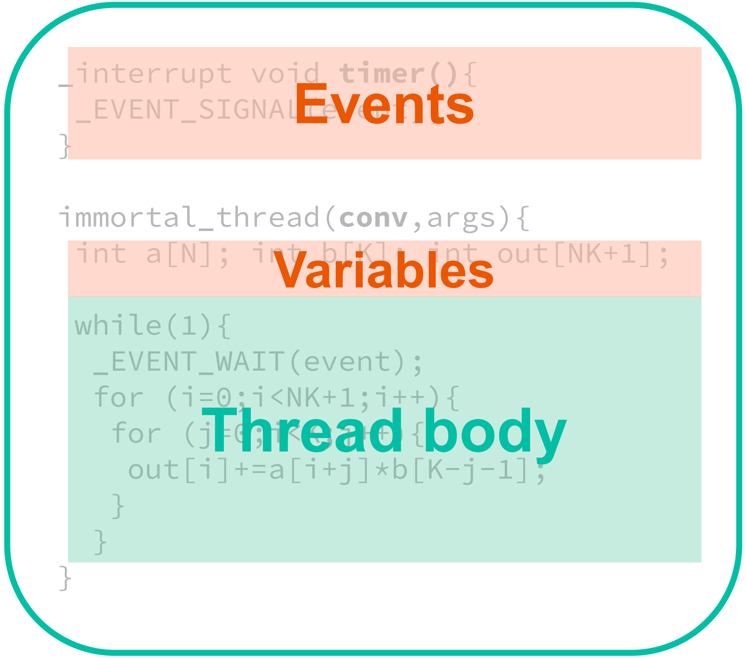
```
_interrupt void timer(){
 _EVENT_SIGNAL(event);
}

immortal_thread(conv,args){
 int a[N]; int b[K]; int out[NK+1];

 while(1){
  _EVENT_WAIT(event);
  for (i=0;i<NK+1;i++){
   for (j=0;i<K;j++){
    out[i]+=a[i+j]*b[K-j-1];
   }
  }
}
```

Immortal Threads

Programmers develop programs in a **multithreaded fashion**

Think only **event-driven aspects**

- identify the events
- threads as event handlers
- manage state management and transitions



Immortal Threads

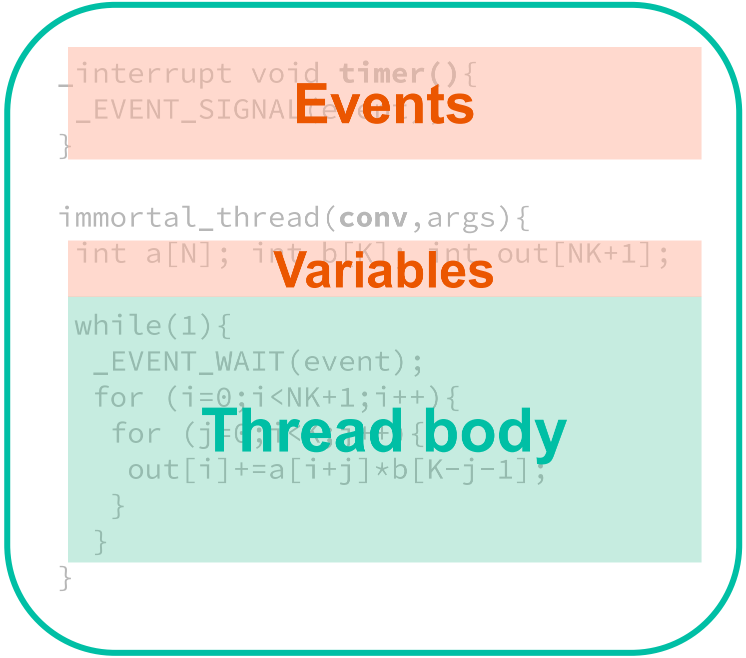
Programmers develop programs in a **multithreaded fashion**

Think only **event-driven aspects**

- identify the events
- threads as event handlers
- manage state management and transitions

Forget about the intermittent execution

no checkpoints + no tasks



Immortal Threads

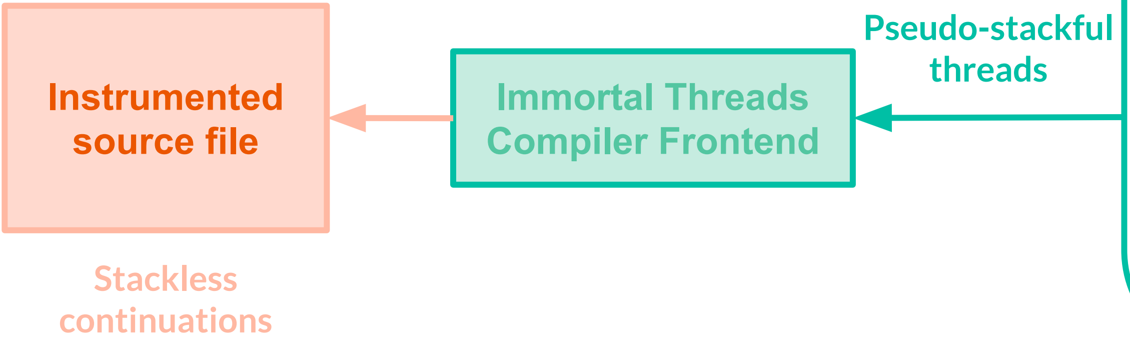
Programmers have the view of programming a *continuously powered system*.

Common *multithreaded event-driven* **language constructs**.

```
_interrupt void timer(){
 _EVENT_SIGNAL(event);
}

immortal_thread(conv,args){
 int a[N]; int b[K]; int out[NK+1];

 while(1){
  _EVENT_WAIT(event);
  for (i=0;i<NK+1;i++){
   for (j=0;i<K;j++){
    out[i]+=a[i+j]*b[K-j-1];
   }
  }
}
```

Immortal Threads

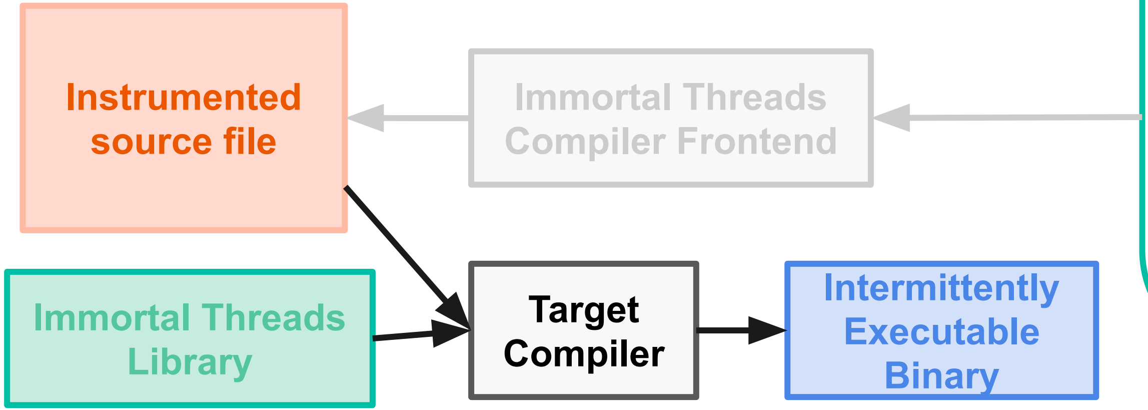
Immortal Threads **compiler frontend** performs a **source-to-source** transformation.




```
_interrupt void timer(){
 _EVENT_SIGNAL(event);
}

immortal_thread(conv,args){
 int a[N]; int b[K]; int out[NK+1];

 while(1){
  _EVENT_WAIT(event);
  for (i=0;i<NK+1;i++){
   for (j=0;i<K;j++){
    out[i]+=a[i+j]*b[K-j-1];
   }
  }
}
```

Immortal Threads

Immortal Threads **compiler frontend** performs a **source-to-source** transformation.




```
_interrupt void timer(){
 _EVENT_SIGNAL(event);
}

immortal_thread(conv,args){
 int a[N]; int b[K]; int out[NK+1];

 while(1){
  _EVENT_WAIT(event);
  for (i=0;i<NK+1;i++){
   for (j=0;i<K;j++){
    out[i]+=a[i+j]*b[K-j-1];
   }
  }
}
```

Outline

- Introduction
- Prior studies & Problems
- Problem Statement
- ImmortalThreads
  - Overview
  - Implementation
- Evaluation
- Conclusion

Immortal Threads - Library and Compiler

Immortal Threads Library

- Standard **C macros** and **preprocessor directives**.
- Functions for system initialization and scheduling operations.

Compiler Frontend

- LLVM & Clang LibTooling
- Uses **macros** defined Immortal Threads library

Compiler Frontend

Programmer Source

```
immortal_thread(th,args){
 int cnt;
 int i;

 i=0;
 while(1){
  _EVENT_WAIT(event);
  cnt++;
  ...
 }
}
```

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  _def int cnt;

  _WR(i,0)
  while(1){
   _EVENT_WAIT(event);
   _WR_SELF(cnt,cnt,1);
   ...
  }
 _end
}
```

Compiler Frontend

Wrap function body using _begin/_end macros

Programmer Source

```
immortal_thread(th,args){
 int cnt;
 int i;

 i=0;
 while(1){
  _EVENT_WAIT(event);
  cnt++;
  ...
 }
}
```

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  _def int cnt;

  _WR(i,0)
  while(1){
   _EVENT_WAIT(event);
   _WR_SELF(cnt,cnt,1);
   ...
  }
 _end
}
```

Compiler Frontend

Instrument all local variables by using **_def** macro

Programmer Source

```
immortal_thread(th,args){
 int cnt;
 int i;

 i=0;
 while(1){
  _EVENT_WAIT(event);
  cnt++;
  ...
 }
}
```

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  _def int cnt;

  _WR(i,0)
  while(1){
   _EVENT_WAIT(event);
   _WR_SELF(cnt,cnt,1);
   ...
  }
 _end
}
```

Compiler Frontend

Instrument all local variables by using **_def** macro

Programmer Source

```
immortal_thread(th,args){
 int cnt;
 int i;

 i=0;
 while(1){
  _EVENT_WAIT(event);
  cnt++;
  ...
 }
}
```

persistent static variables with *local scope*

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  _def int cnt;

  _WR(i,0)
  while(1){
   _EVENT_WAIT(event);
   _WR_SELF(cnt,cnt,1);
   ...
  }
 _end
}
```

Compiler Frontend

Variable manipulations using _WR and _WR_SELF

Programmer Source

```
immortal_thread(th,args){
 int cnt;
 int i;

 i=0;
 while(1){
  _EVENT_WAIT(event);
  cnt++;
  ...
 }
}
```

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  _def int cnt;

  _WR(i,0)
  while(1){
   _EVENT_WAIT(event);
   _WR_SELF(cnt,cnt,1);
   ...
  }
 _end
}
```

Compiler Frontend

Programmer Source

```
immortal_thread(th,args){
 int cnt;
 int i;

 i=0;
 while(1){
  _EVENT_WAIT(event);
  cnt++;
  ...
 }
}
```

Power Failure

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  _def int cnt;

  _WR(i,0)
  while(1){
   _EVENT_WAIT(event);
   _WR_SELF(cnt,cnt,1);
   ...
  }
 _end
}
```

Compiler Frontend

Programmer Source

```
immortal_thread(th,args){
 int cnt;
 int i;

 i=0;
 while(1){
  _EVENT_WAIT(event);
  cnt++;
  ...
 }
}
```

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  _def int cnt;

  _WR(i,0)
  while(1){
   _EVENT_WAIT(event);
   _WR_SELF(cnt,cnt,1);
   ...
  }
 _end
}
```

Enabling Micro Continuations

Almost Free Checkpoints

Saves **only** the **program counter** rather than all registers and memory.

Just-in-time Privatization

Creates **private copies** of variables **dynamically** to keep non-volatile memory consistent.

Just 2 Bytes for checkpoints!

Just 8 Bytes for versioning!

Enabling Micro Continuations

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  ...
  _WR(i,0)
  ...
 _end
}
```

After C Preprocessor

```
static _fram priv_buf_t _priv_buf;
void *th(void *args){
 static _fram imm_func_t this;
 switch(this.pc){
      case 0:
       static _fram int i;
       ...
       this.pc = __COUNTER__+1;
      case __COUNTER__:
       i=0;
       ...
 }
}
```

Enabling Micro Continuations

privatization buffer in non-volatile memory

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  ...
  _WR(i,0)
  ...
 _end
}
```

After C Preprocessor

```
static _fram priv_buf_t _priv_buf;
void *th(void *args){
 static _fram imm_func_t this;
 switch(this.pc){
      case 0:
       static _fram int i;
       ...
       this.pc = __COUNTER__+1;
      case __COUNTER__:
       i=0:
       ...
 }
}
```

thread structure in non-volatile memory that holds pc

Enabling Micro Continuations

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  ...
  _WR(i,0)
  ...
 _end
}
```

After C Preprocessor

```
static _fram priv_buf_t _priv_buf;
void *th(void *args){
 static _fram imm_func_t this;
 switch(this.pc){
      case 0:
       static _fram int i;
       ...
       this.pc = __COUNTER__+1;
      case __COUNTER__:
       i=0;
       ...
 }
}
```

Enabling Micro Continuations

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  ...
  _WR(i,0)
  ...
 _end
}
```

After C Preprocessor

```
static _fram priv_buf_t _priv_buf;
void *th(void *args){
 static _fram imm_func_t this;
 switch(this.pc){
     case 0:
      static _fram int i;
      ...
      this.pc = __COUNTER__+1;
     case __COUNTER__:
      i=0;
      ...
 }
}
```

Enabling Micro Continuations

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  ...
  _WR(i,0)
  ...
 _end
}
```

After C Preprocessor

```
static _fram priv_buf_t _priv_buf;
void *th(void *args){
 static _fram imm_func_t this;
 switch(this.pc){
      case 0:
       static _fram int i;
       ...
       this.pc = __COUNTER__+1;
      case __COUNTER__:
       i=0;
       ...
 }
}
```

Enabling Micro Continuations

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  ...
  _WR(i,0)
  ...
 _end
}
```

After C Preprocessor

```
static _fram priv_buf_t _priv_buf;
void *th(void *args){
 static _fram imm_func_t this;
 switch(this.pc){
      case 0:
       static _fram int i;
       ...
       this.pc = __COUNTER__+1;
      case __COUNTER__:
       i=0;
       ...
 }
}
```

Enabling Micro Continuations

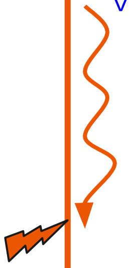
After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  ...
  _WR(i,0)
  ...
 _end
}
```

After C Preprocessor

```
static _fram priv_buf_t _priv_buf;
void *th(void *args){
 static _fram imm_func_t this;
 switch(this.pc){
      case 0:
       static _fram int i;
       ...
       this.pc = __COUNTER__+1;
      case __COUNTER__:
       i=0:
       ...
 }
}
```

Power Failure



Enabling Micro Continuations

After Compiler Pass

```
immortal_thread(th,args){
 _begin
  _def int i;
  ...
  _WR(i,0)
  ...
 _end
}
```

After C Preprocessor

```
static _fram priv_buf_t _priv_buf;
void *th(void *args){
 static _fram imm_func_t this;
 switch(this.pc){
      case 0:
       static _fram int i;
       ...
       this.pc = __COUNTER__+1;
      case __COUNTER__:
       i=0:
       ...
 }
}
```

Continue from the last case statement

Almost Free Checkpoints

Checkpoint Macro

```
#define _CP() \
 this.pc = __COUNTER__ + 1; \
 case __COUNTER__:
```

Almost Free Checkpoints

Checkpoint Macro

```
#define _CP() \
 this.pc = __COUNTER__ + 1; \
 case __COUNTER__:
```

WRITE Macro

```
#define _WR(arg,val)  \
 _CP();               \
 arg=val;
```

Almost Free Checkpoints

Checkpoint Macro

```
#define _CP() \
 this.pc = __COUNTER__ + 1; \
 case __COUNTER__:
```

WRITE Macro

```
#define _WR(arg,val)  \
 _CP();               \
 arg=val;
```

```
...
x=y;
y=z;
...
```

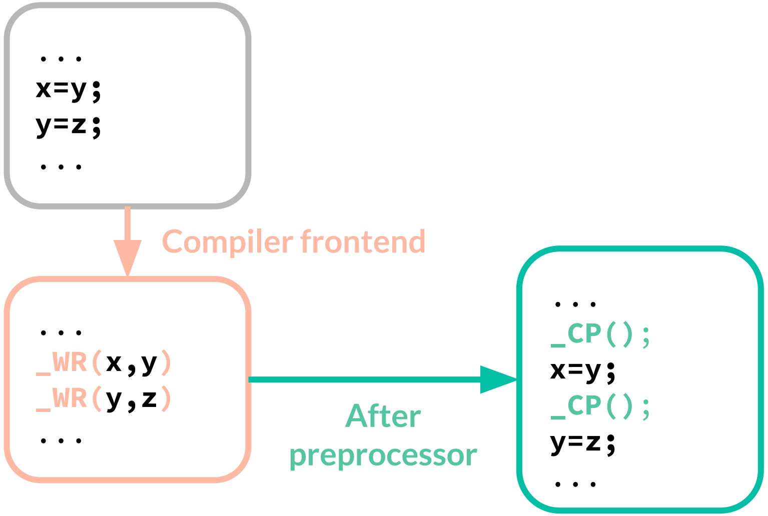
Almost Free Checkpoints

Checkpoint Macro

```
#define _CP() \
 this.pc = __COUNTER__ + 1; \
 case __COUNTER__:
```

WRITE Macro

```
#define _WR(arg,val)  \
 _CP();               \
 arg=val;
```



Just-in-Time Privatization

Single memory updates that include WAR dependency

```
...
x++
...
```

Just-in-Time Privatization

WRITE_SELF Macro

```
#define _WR_SELF(arg,val)          \
 _CP();                            \
 priv_buf=val;                     \
 _CP();                            \
 arg = priv_buf;
```

Single memory updates that include WAR dependency

```
...
x++
...
```

Require **two-phase commit**

Just-in-Time Privatization

WRITE_SELF Macro

```
#define _WR_SELF(arg,val)          \
 _CP();                            \
 priv_buf=val;                     \
 _CP();                            \
 arg = priv_buf;
```

Single memory updates that include WAR dependency

```
...
x++
...
```

Require **two-phase commit**

Just-in-Time Privatization

WRITE_SELF Macro

```
#define _WR_SELF(arg,val)       \
 _CP();                         \
 priv_buf=val;                  \
 _CP();                         \
 arg = priv_buf;
```

Single memory updates that include WAR dependency

```
...
x++
...
```

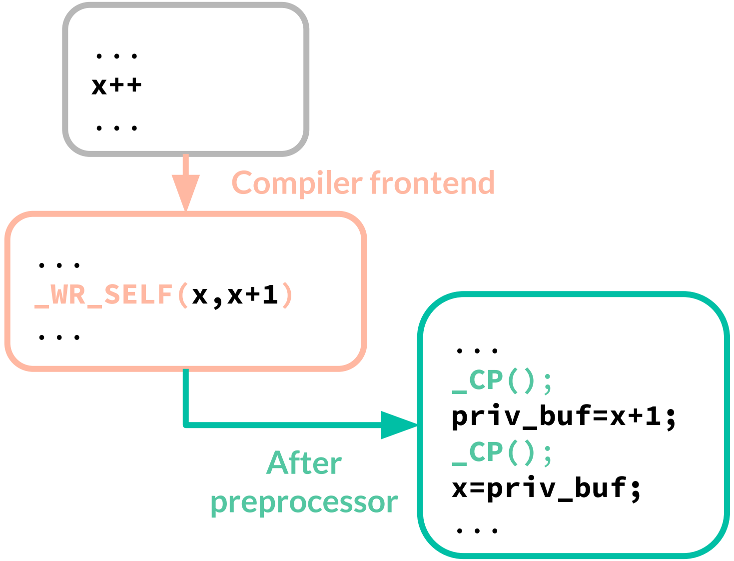
Require **two-phase commit**

Just-in-Time Privatization

WRITE_SELF Macro

```
#define _WR_SELF(arg,val)          \
 _CP();                            \
 priv_buf=val;                     \
 _CP();                            \
 arg = priv_buf;
```

Single memory updates that include WAR dependency



- no compiler analysis to detect idempotent code blocks
- no need for static versioning

Thread Scheduling

Immortal Threads implements **round-robin** scheduling.

Thread 1

```
immortal_thread(th1,args){
 int x;
 int y;
 ...
 x = 5;
 ...
 y = x;
 ...
 _SEM_POST(sem);
 ...
}
```

Thread 2

```
immortal_thread(th1,args){
 int z;
 ...
 z = 5;
 ...
 _SEM_WAIT(sem);
  ...
}
```

Thread Scheduling

Immortal Threads implements **round-robin** scheduling.

Thread 1

```
immortal_thread(th1,args){
 int x;
 int y;
 ...
 x = 5;
 ...
 y = x;
 ...
 _SEM_POST(sem);
 ...
}
```

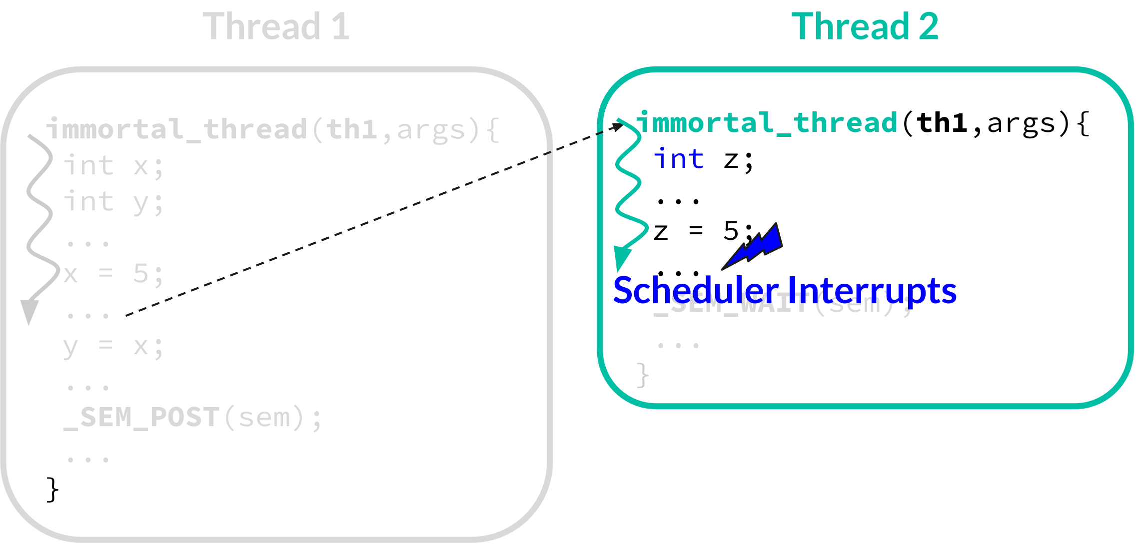
Scheduler Interrupts

Thread 2

```
immortal_thread(th1,args){
 int z;
 ...
 z = 5;
 ...
 _SEM_WAIT(sem);
  ...
}
```

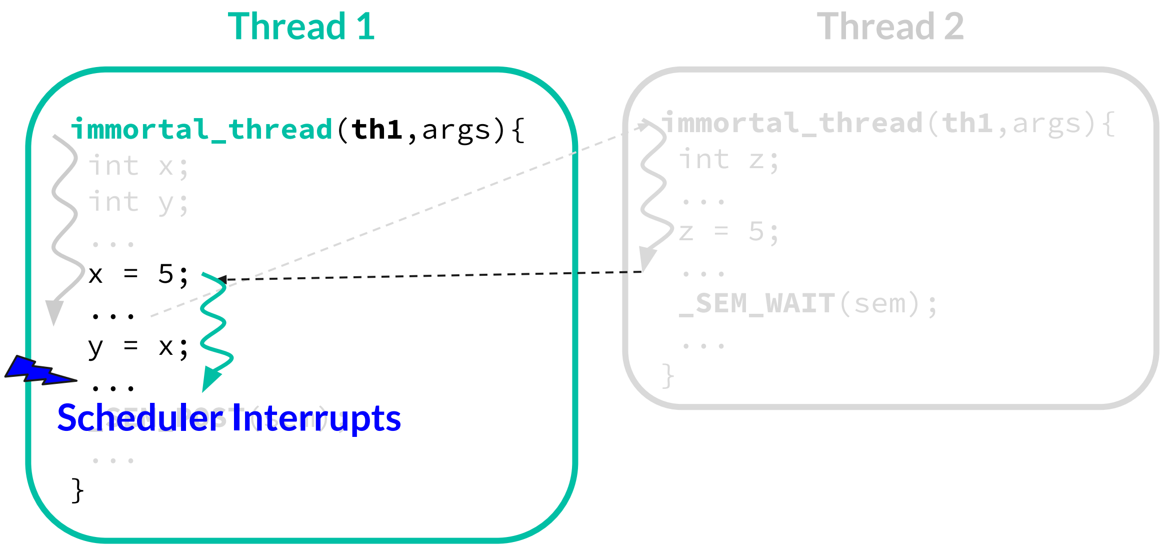
Thread Scheduling

Immortal Threads implements **round-robin** scheduling.



Thread Scheduling

Immortal Threads implements **round-robin** scheduling.



For More Details...

See our paper!

- Compiler front-end
- Function calls and sharing
- Scheduling Details
- Semaphores, Mutexes
- …

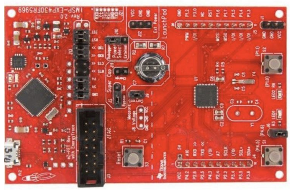
Outline

- Introduction
- Prior studies & Problems
- Problem Statement
- ImmortalThreads
  - Overview
  - Implementation
- Evaluation
- Conclusion

Evaluation

1 Testbed Setup

Powercast RF Energy Harvester



MSP430FR5994 with 1 MHz CPU Speed

2 Runtime Systems

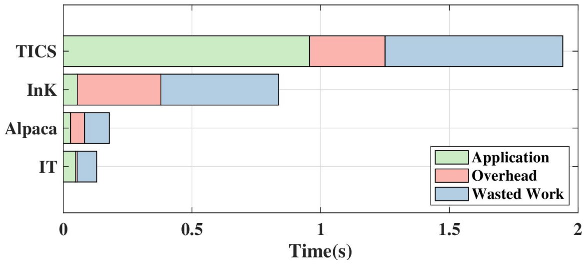
- **Alpaca** [OOPSLA'17] (task-based)
- **InK** [SenSys'18] (task-based)
- **TICS** [ASPLOS'20] (checkpoints)

Applications

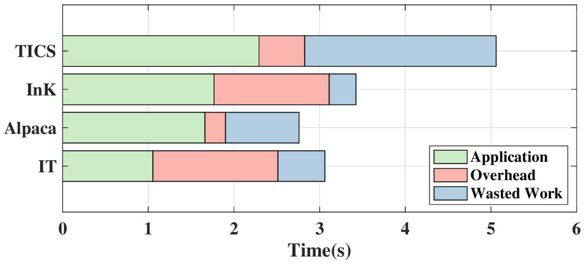
- Bitcount
- Activity Recognition
- Cuckoo Filter
- Deep Neural Network

Evaluation - Benchmarks

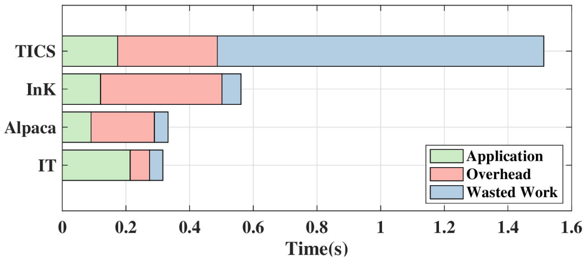
Bitcount



Activity Recognition



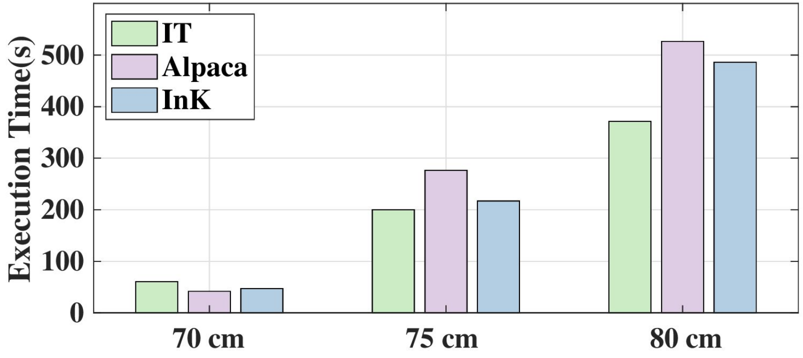
Cuckoo Filter



Immortal Threads reduced **wasted work** and **throughput**.

Evaluation - Deep Neural Network

InK and Alpaca uses loop continuation (**violates** the task-based model)



1mF

Thanks to the **micro-continuations**, Immortal Threads becomes **superior** as the **power failure rate increases.**

Summary of Evaluations

Factors Effecting the Performance

- Application’s memory access patterns
- Frequency of the power failures.

For more results, see our paper!

- Code Size, Memory Overheads
- Monitoring Application
- …

Outline

- Introduction
- Prior studies & Problems
- Problem Statement
- ImmortalThreads
  - Overview
  - Implementation
- Evaluation
- Conclusion

Conclusions - Immortal Threads

- Enables **pseudo-stackful** multithreaded programming.
- Brings the missing **event-driven primitives**
- Removes **cognitive burden** of intermittent computing

All these features come with a **comparable overhead**.

https://tinysystems.github.io/ImmortalThreads

Immortal Threads: Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers

Eren Yildiz[2], Lijun Chen[1], **Kasim Sinan Yildirim**[1]

[1]University of Trento, Italy, [2]Ege University, Turkey

16th USENIX Symposium on Operating Systems Design and Implementation
July 11–13, 2022, Carlsbad, CA, USA