Constraint Based Compiler Optimization for Energy Harvesting Applications

Anonymous author
Anonymous affiliation

Abstract

We propose a method for optimizing the energy efficiency of software code running on small computing devices in the Internet of Things (IoT) that are powered exclusively by electricity harvested from ambient energy in the environment. Due to the weak and unstable nature of the energy source, it is challenging for developers to manually optimize the software code to deal with mismatch between the intermittent power supply and the computation demand. Our method overcomes the challenge by using a combination of three techniques. First, we use static program analysis to automatically identify opportunities for *precomputation*, i.e., computation that may be performed ahead of time as opposed to just in time. Second, we optimize the precomputation policy, i.e., a way to split and reorder steps of a computation task in the original software to match the intermittent power supply while satisfying a variety of system requirements; this is accomplished by formulating energy optimization as a constraint satisfiability problem and then solving the problem using an off-the-shelf SMT solver. Third, we use a state-of-the-art compiler platform (LLVM) to automate the program transformation to ensure that the optimized software code is correct by construction. We have evaluated our method on a large number of benchmark programs, which are C programs implementing secure communication protocols that are popular for energy-harvesting IoT devices. Our experimental results show that the method is efficient in optimizing all benchmark programs. Furthermore, the optimized programs significantly outperform the original programs in terms of both energy efficiency and latency, and the overall improvement ranges from 2X to 29X.

2012 ACM Subject Classification Software engineering and Programming languages

Keywords and phrases Compiler optimization; Static analysis; Constraint solving; Energy harvesting; Internet of Things

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Energy harvesting is an environment-friendly technology that converts ambient energy in the environment such as sunlight, RF emission, and vibration into electricity [35, 33, 12, 29, 31, 28, 42]. When being used to power small computing devices in the Internet of Things (IoT), it avoids a main problem in the deployment of IoT at scale, which is the need to frequently change batteries [6]. Due to this reason, energy harvesting has been increasingly used in real-world deployment of IoT devices [38, 26]. However, due to the weak and unstable nature of the energy source, it is often challenging for developers to manually optimize the software code running on these IoT devices, to deal with problems caused by mismatch between the intermittent power supply and the often unpredictable computation demand.

Consider an IoT device powered by electricity harvested from sunlight as an example. During the day time, there may be significantly more harvested electricity than the combined total of what can be stored in the supercapacitor of the device, and what can be consumed by the software tasks running on the device. During the night time, however, the electricity stored in the supercapacitor may be significantly less than what is needed by the software tasks running on the device. In this context, an important research question is whether the *mismatch between supply and demand* can be avoided, or at the very least mitigated by rewriting the original software in such a way that, while the functionality of the software

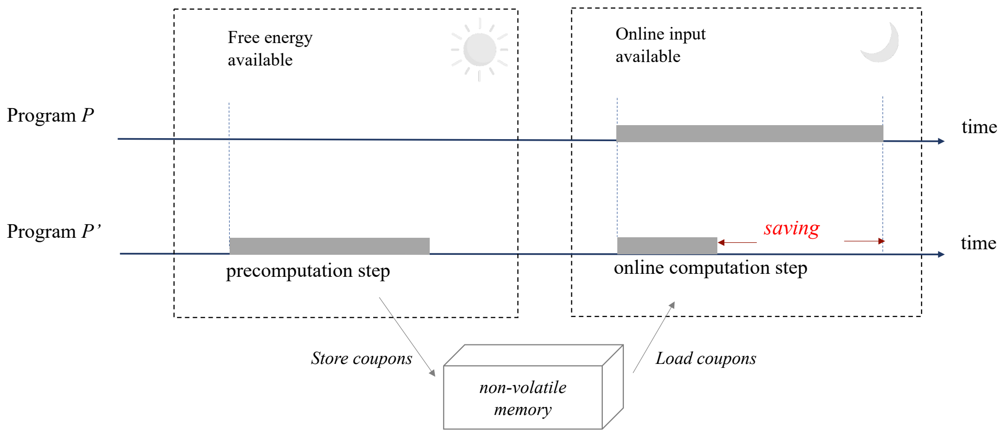


Figure 1 Using precomputation to reduce the energy needed by the (online) computation task.

remains the same, the overall energy efficiency is improved. Prior work [10, 11, 41] has demonstrated the feasibility of this approach, based on two observations made for typical energy-harvesting IoT devices.

The first observation is that, since the software tasks on an IoT device are only executed from time to time, rather than continuously, the device may be idle when ambient energy in the environment is abundant (e.g., sunlight during the day time) and yet the supercapacitor used to store the harvested electricity is full. In such a case, the *freely available* energy in the environment cannot be utilized. The second observation is that, in such an IoT device, the most common computation tasks are collecting sensor data from time to time, and encrypting these sensor data before sending them to some remote servers, e.g., servers in the cloud. Thus, the most time-consuming and energy-consuming part of the computation is the execution of the secure communication protocol. While the sensor data may have to be collected *just in time*, a significant part of the secure communication protocol (e.g., computing security tokens needed for encrypting the sensor data) may be executed *ahead of time*. This leads to the idea of leveraging the *precomputation* opportunities to utilize the *freely-available* ambient energy in the environment.

Figure 1 illustrates an example of using the idea to optimize a computation task that must be executed during the night time, when ambient energy in the environment is not available. While the original program (P) has to execute the entire computation task during the night time using electricity stored in the supercapacitor, the optimized program (P') executes a significant part of the task during the day time, by harvesting the freely-available ambient energy that otherwise would have to be wasted due to the storage limit of the supercapacitor. In some sense, the precomputation performed during the day time transforms the solar energy to a digital form, called *coupons*, and stores them in the non-volatile memory of the IoT device. During the night time, these coupons are used to lower the energy cost of the remaining part of the computation task.

There are two main benefits. The first one is reduction in latency for the online computation part, since a significant portion of the computation task has been completed ahead of time. The second one is increase in the number of computation tasks that can be completed by the device. As a concrete example, Suslowicz et al. [41] show that, for a popular secure communication protocol based on *one-time pad (OTP)* [39], using precomputed OTPs for

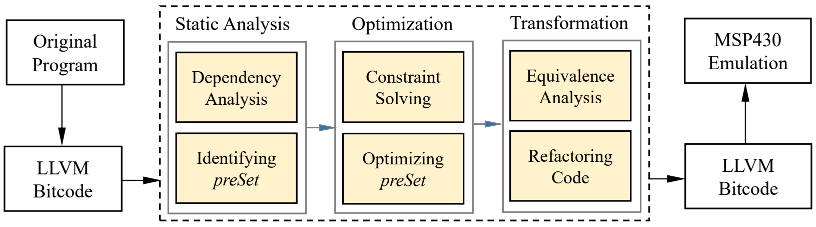


Figure 2 The overall flow of our constraint based method for energy optimization.

sensor data encryption reduces the energy cost of the online computation to 3.55% of the original energy cost needed for AES-OFB. Since the energy used to precompute OTPs is free, the overall energy reduction is close to 28 times (28X). To understand what this means, consider an IoT device that must complete 20 tasks during the day time and 20 tasks during the night time, but the electricity stored in the supercapacitor is only enough to support the completion of 2 tasks during the night time. Without precomputation, the device may be able to complete 20 tasks during the day time and only 2 tasks during the night time. By leveraging the coupons precomputed during the day time, the same device is able to complete 20 tasks during the day time and 20 tasks during the night time.

However, to obtain the aforementioned benefits of precomputation, the current state of the art [10, 11, 41] requires a domain expert to optimize the software code manually, which is not only labor intensive but also error prone. Furthermore, the domain expert must be familiar with both the functionality of the software code and the energy characteristics of the hardware platform. The domain expert must also consider all of the system requirements while making the trade-off between energy reduction and increase in storage cost. In addition, manual optimization does not respond well to frequent software updates in practice: if the original software code is updated due to a bug fix or a security patch, there will be no easy way to update the manually-optimized software code.

To solve these problems, we propose a fully automated method for optimizing the energy efficiency of software running on energy-harvesting IoT devices. Toward this end, we must overcome three technical challenges. The first challenge is to identify the precomputation opportunities from the original software code automatically. The second challenge is to optimize the precomputation policy by exploiting the *energy–storage* trade-off and deciding which part of the computation task should be precomputed and which part of the computation task should be computed just in time. The third challenge is to automatically transform the software code to implement the energy optimization policy.

Figure 2 shows the overall flow of our method, which builds upon the state-of-the-art LLVM compiler platform [27]. Given the original program, our method takes three steps to produce the optimized program. In the first step, our method conducts a static analysis of the original program to identify precomputation opportunities, which are captured by *preSet* — the set of instructions in the program that may be computed ahead of time. In the second step, our method computes an optimal subset of *preSet* based on a variety of system requirements, to minimize the energy cost while satisfying all requirements, including the storage limit of non-volatile memory used to save precomputation results. In the third step, our method leverages the LLVM compiler to generate the optimized program that has the ability to load the precomputation results from non-volatile memory and leverage them to speed up the just-in-time (online) computation part of the task. Finally, we evaluate the performance of the optimized programs on a popular hardware platform (MSP430 [24]) for energy-harvesting applications.

At the center of our method is a constraint based technique for optimizing the precomputation policy. The policy captures a solution to the complex optimization problem. The optimization problem is complex for several reasons. First, just because an instruction may be precomputed does not mean it is always beneficial to precompute it, since precomputing does not always reduce energy cost; there is a trade-off between the cost of storing a precomputed coupon and the benefit of avoiding computing it directly. Second, decisions on *which instructions to precompute* cannot be made in isolation, since many of these instructions are dependent of each other; the precomputation policy has to consider all of the intra- and inter-procedural control- and data-flow dependencies in the program. Third, the size of the non-volatile memory used to store the precomputed coupons may not grow monotonically with the number of precomputed instructions, and furthermore, not all intermediate computation results in the program need to be stored as coupons in non-volatile memory. We will use concrete examples Section 2 to illustrate these challenges and our proposed solution to overcome these challenges.

To demonstrate the effectiveness of our method, we have implemented it as a software tool and evaluated the tool on a large number of benchmark programs. Our tool builds upon the state-of-the-art LLVM compiler platform [27] and the Z3 SMT solver [16]. Specifically, we use LLVM to parse the original software code (written in the C language), conduct static program analysis, and generate the optimized software code; we use Z3 to solve the constraint satisfiability problems formulated by our method. Our tool was evaluated on 26 benchmark programs, which are C programs implementing popular secure communication protocols for IoT devices; in total, they have 31,113 lines of C code. Our target hardware platform is MSP430 [24], a family of ultra-low-power microcontroller units (MCUs) popular for energy-harvesting IoT applications.

Our experimental results are promising. In terms of the efficiency of our method, the experimental evaluation shows that all of the benchmark programs can be optimized by our tool quickly, and the optimization time is always limited to a few seconds. In terms of the effectiveness of our method, the experimental evaluation shows that all of the optimized programs significantly outperform the original programs in terms of both energy efficiency and latency. Specifically, reduction in the overall energy cost ranges from 2X to 29X.

To summarize, this paper makes the following contributions:

- We propose a compiler based technique for automatically identifying precomputation opportunities in the software code using static analysis and then exploiting these opportunities using a semantic-preserving program transformation.
- We formulate energy optimization as a constraint satisfiability problem and solve the problem using an off-the-shelf SMT solver; this approach is not only flexible but also efficient in minimizing the energy cost while satisfying a variety of system requirements.
- We implement the method using a state-of-the-art compiler (LLVM) and a popular hardware platform (MSP430) for energy-harvesting applications, and demonstrate the effectiveness on a large number of benchmark programs.

The remainder of this paper is organized as follows. First, we provide the technical background in Section 2. Then, we present our top-level procedure in Section 3. This is followed by our method for identifying the precomputation opportunities in Section 4, optimizing the precomputation policy in Section 5, and transforming the software code in Section 6. We present our experimental results in Section 7. We review the related work in Section 8 and, finally, give our conclusion in Section 9.

2 Background

In this section, we review the technical background, including the characteristics of the hardware platform (MSP430) for energy-harvesting applications, and an example software program to motivate our approach.

2.1 The Hardware Platform

MSP430 is a family of microcontroller units (MCUs) based on a 16-bit RISC instruction set architecture. Due to our focus on energy-harvesting applications, we are concerned with MSP430 MCUs that have the main memory partitioned into the volatile part and the non-volatile part. Depending on the application, the data may be stored either in volatile memory or in non-volatile memory. These MCUs have a large number of configuration parameters, including sixteen nominal frequencies in the range 0.06 MHz to 16 MHz. For example, they may run in a low-power mode at the clock frequency of 1 MHz and the supply voltage of 1.8V, or in a high-performance mode at the clock frequency of 16 MHz and the supply voltage of 2.9V.

Since MSP430 MCUs are designed for low-power applications, they have no instruction cache or data cache. Unlike high-end CPUs widely used in servers and desktops, which routinely use advanced frequency or voltage scaling techniques, low-power MCUs such as MSP430 have significantly simpler energy models: fluctuations in the power consumption are primarily due to the dynamics in supply voltage and clock speed. In fact, power consumption may be modeled using a non-linear function derived by empirically measuring the impact of varying voltage supplies and clock speeds on the power consumption of real hardware for all possible MCU configurations [8].

Accurate compile-time analysis for energy prediction [15, 9] is well studied topic for transiently powered computing systems [8], where software developers need to know the worst-case energy cost of a computation task, to maximize the software's utilization of the electricity harvested from the environment and to ensure timely checkpointing of the program state before loss of power. The accuracy of such compile-time analysis techniques have come close to direct hardware emulation. While direct hardware emulation [21, 13] offers the highest possible accuracy due to the direct measurement on target hardware, it does not offer the level of convenience and automation desired at the early stages of software development.

In this work, we evaluate the performance of our proposed method using MSPSim [17, 34], which is a popular compile-time analysis tool for MSP430, Specifically, we use MSPSim to compute and then compare the latency and energy cost of all benchmark programs, before and after applying our constraint-based optimization technique.

MSPSim allows the software developer to tag a piece of the software code for which energy consumption will be estimated. It does this by first generating the assembly code for MSP430, and then analyzing the assembly code to compute the number of MCU cycles needed to execute each basic block. Then, it estimates the energy consumption of each basic block based on the empirically derived energy model, the supply voltage, and the clock speed of the device.

At a high level, the energy consumption depends on the supply voltage as well as the electrical current for a given resistance of the MCU, the latter of which in turn depends on the supply voltage and the clock speed. For more details on the energy model used in such compile-time analysis tools, refer to Ahmed et al. [8].

```
 1  __interrupt void ISR(void) {
 2      if (msg_ready) {
 3              wots(msg, pub_key, sig);
 4              //Send the pair <pub_key,sig> to verifier;
 5      }
 6  }
 7  void wots(MSG msg, KEY pub_key, SIG sig) {
 8      gen_key(priv_key, pub_key);
 9      sign(msg, priv_key, sig);
10  }
```

Figure 3 An example program that invokes the W-OTS routine when `msg` is ready.

2.2 The Software Program

We use an example software program written for MSP430 to motivate our work.

Figure 3 shows the program, where `ISR` stands for the interrupt service routine. Assume that the routine is triggered periodically by a timer. Whenever the input data stored in `msg` is ready, the subroutine `wots()` is invoked (Line 3). It implements a hash-based cryptographic primitive called the *Winternitz* one-time signature (W-OTS [4]). Here, `msg` is the input, while `pub_key` and `sig` are the output. After generating the output, the device sends the pair (`pub_key,sig`) to a verifier on a remote server (Line 4).

Let us take a closer look at the routine `wots()` defined in Lines 7-10, which consists of two subroutines. The subroutine `gen_key()` is invoked first, which returns the private key `priv_key` and the public key `pub_key` as output. Then, the subroutine `sign()` is invoked, which takes `msg` and `priv_key` as input and returns the signature `sig` as output.

Since the input `msg` may be sensor data generated by the system *just in time*, in the context of our work, it is called an *online* input. Furthermore, any output or intermediate variable that is control- or data-dependent of the *online* input must be computed *just in time*. In contrast, results that do not depend on the *online* input may be computed *ahead of time*.

2.2.1 The Original Program

Figure 4 shows the definitions of the two subroutines invoked by `wots()`.

The subroutine `sign()` in Line 7 takes `msg` and `priv_key` as input and returns `sig` as output. While `msg` is an *online* input, `priv_key` is computed by the subroutine `gen_key()`. In this sense, `sign()` depends on `gen_key()`.

The subroutine `gen_key()` does not have any input, and thus does not depend on any other subroutine. More importantly, it does not depend on any *online* input. Thus, `gen_key()` may be executed ahead of time, e.g., whenever ambient energy is available to the harvester. It means that both `priv_key` and `pub_key` may be computed ahead of time. These precomputed keys may be saved to non-volatile memory as *coupons*, and later used by `sign()` to encrypt the *online* input `msg`.

Although the subroutine `sign()` partially depends on the *online* input `msg`, and thus cannot be executed ahead of time *in its entirety*, a significant part of the function body can still be executed ahead of time. Specifically, the subroutine `gen_random()` does not depend on the *online* input at all, and the subroutine `memcpy()` depends only on `rand` computed by `gen_random()`; thus, both subroutines can be computed ahead of time.

If we continue this analysis by going down the chain of function calls, we may identify more precomputation opportunities, e.g., instructions inside subroutines `message_digest()`

```
gen_key(priv_key, pub_key) {
    gen_random(priv_key, PRIV_KEY_SIZE);
    sha256_init(&keyHash);
    sha256_update(&keyHash, priv_key, PRIV_KEY_SIZE);
    sha256_final(&keyHash, pub_key);
}
sign(msg, priv_key, sig) {
    gen_random(rand, SHA_BLK_SIZE);
    memcpy(sig, rand, SHA_BLK_SIZE);
    message_digest(digest_bits, sig, msg);
    gen_sig(sig, priv_key, digest_bits);
}
```

Figure 4 Definitions of the subroutines used by the W-OTS routine.

```
1   wots_precom(msg, pub_key, sig) {
2       gen_key(priv_key, pub_key);
3       //NVM-Store <priv_key, pub_key> to coupon pool;
4       sign_precom(msg, priv_key, sig);
5   }
6   wots_online(msg, pub_key, sig) {
7       //NVM-Load <priv_key, pub_key> from coupon pool;
8       sign_online(msg, priv_key, sig);
9   }
10  gen_key(priv_key, pub_key) {
11      gen_random(priv_key, PRIV_KEY_SIZE);
12      sha256_init(&keyHash);
13      sha256_update(&keyHash, priv_key, PRIV_KEY_SIZE);
14      sha256_final(&keyHash, pub_key);
15  }
16  sign_precom(msg, priv_key, sig) {
17      gen_random(rand, SHA_BLK_SIZE);
18      memcpy(sig, rand, SHA_BLK_SIZE);
19      //NVM-Store <sig> to coupon pool;
20  }
21  sign_online(msg, priv_key, sig) {
22      //NVM-Load <sig> from coupon pool;
23      message_digest(digest_bits, sig, msg);
24      gen_sig(sig, priv_key, digest_bits);
25  }
```

Figure 5 Conceptually, the program may be divided into two parts (*precom* and *online*).

and `gen_sig()`. In our proposed method, this process of systematically identifying these precomputation opportunities is automated, based on static program analysis techniques.

2.2.2 Dividing into Two Parts

Based on the precomputation opportunities identified by static program analysis, the original program may be divided into two parts: the precomputation (*precom*) part and the online computation (*online*) part, as shown by Figure 5.

Specifically, the top-level routine `wots()` is divided into `wots_precom()` and `wots_online()`. The subroutine `wots_precom()` may be invoked ahead of time, since it does not depend on the *online* input `msg` at all. After invoking `gen_key()` to compute the public and private keys, denoted `priv_key` and `pub_key`, it stores them in non-volatile memory (Line 3). Then, it invokes `sign_precom()` defined in Line 16, to compute the signature `sig`, before storing it in non-volatile memory (Line 19).

The subroutine `wots_online()` must be invoked just in time, since it depends on the *online* input `msg`. This subroutine first loads the precomputed keys `priv_key` and `pub_key` from non-volatile memory (Line 7) and then invokes `sign_online()` defined in Line 21. Inside `sign_online()`, the precomputed signature `sig` is loaded from non-volatile memory (Line 22) and then used together with `msg` and `priv_key` to compute the final version of the signature `sig` (Lines 23-24).

According to our experimental evaluation (to be presented in Section 7), on low-power devices such as MSP430, this kind of precomputation can reduce the energy cost of running W-OTS to 33.27% of the original cost. In other words, it is more than 3X reduction. Thus, with the same amount of electricity used to run the original W-OTS program once, now, we can run the optimized W-OTS program three times.

2.2.3 Challenges in Optimization

However, just because an instruction in the program may be precomputed, e.g., since it does not depend on any *online* input, does not mean that it is always beneficial to precompute it, since precomputation does not always reduce the overall energy cost. Depending on the hardware platform, it is possible for the cost of storing and retrieving the precomputed result to outweigh the benefit.

For example, in Line 18 of Figure 5, if we choose to precompute `memcpy()` inside the subroutine `sign()`, the energy cost of loading the precomputed coupon `sig` from non-volatile memory may be slightly higher than the energy needed to execute `memcpy()` directly. If that is the case, precomputation should be avoided.

In general, whether precomputation is beneficial or not depends on the software code as well as the hardware platform. Consider the characteristics of volatile and non-volatile memory used in MSP430FR5969 [24] as an example. According to the hardware data-sheet, at the clock frequency of 8 MHz, the energy per clock cycle is 0.33 nJ for volatile memory, but is 0.42 nJ for non-volatile memory. This kind of information must be considered during the optimization.

Furthermore, decisions on which instructions to precompute cannot be made in isolation, since many of these instructions are dependent of each other according to the control and data flows of the program. Therefore, we must consider all of the intra- and inter-procedural control- and data-flow dependencies in the program while performing the optimization.

These are the reasons why we propose the constraint based method. By first formulating it as a constraint satisfiability problem and then solving the problem using an off-the-shelf SMT solver, we are able to optimally partition the original program into the precomputation part and the online computation part, while satisfying a variety of requirements coming from the hardware platform as well as the software program.

2.2.4 The Optimized Program

To keep the size of the optimized program small, we do not actually divide the program into two parts as shown by Figure 5. Instead, we keep the two parts in a single program, and try to retain the original control and data flows of the program as much as possible.

Figure 6 illustrate our method by showing the optimized program for the original program in Figure 4. Our method adds two parameters, `precom_flag` and `online_flag`, to represent the following three use cases:

- When ⟨ `precom_flag,online_flag` ⟩ = ⟨ `true,false` ⟩, it does precomputation.
- When ⟨ `precom_flag,online_flag` ⟩ = ⟨ `false,true` ⟩, it does online computation.

```
wots_trans(msg, pub_key, sig, precom_flag, online_flag) {
  if (precom_flag == true)
      gen_key(priv_key, pub_key);
  if (online_flag != true)
      //NVM-Store <priv_key, pub_key> to coupon pool;
  if (precom_flag != true)
      //NVM-Load <priv_key, pub_key> from coupon pool;
  sign_trans(msg, priv_key, sig, precom_flag, online_flag);
}
sign_trans(msg, priv_key, sig, precom_flag, online_flag) {
  if (precom_flag == true) {
      gen_random(rand, SHA_BLK_SIZE);
      memcpy(sig, rand, SHA_BLK_SIZE);
  }
  if (online_flag != true)
      //NVM-Store <sig> to coupon pool;
  if (precom_flag != true)
      //NVM-Load <sig> from coupon pool;
  if (online_flag == true) {
      message_digest(digest_bits, sig, msg);
      gen_sig(sig, priv_key, digest_bits);
  }
}
```

Figure 6 Merging the two parts into a single optimized W-OTS routine.

```
1  __interrupt void ISR(void) {
2      if(!msg_ready}
3        if (ambient_energy_available)
4            wots_trans(NULL, pub_key, sig, true, false); //precom (part 1)
5      }
6      else {
7        if (!ambient_energy_available)
8            wots_trans(msg, pub_key, sig, false, true); //online (part 2)
9        else
10           wots_trans(msg, pub_key, sig, true, true); //combined (part 1 + part 2)
11       //Send the pair <pub_key,sig> to verifier;
12     }
13 }
```

Figure 7 Different scenarios for invoking the optimized W-OTS routine.

- When ⟨ `precom_flag,online_flag` ⟩ = ⟨ `true,true` ⟩, it acts as the original program.

Compared to the original program in Figure 4, the only difference in Figure 6 is the addition of two flags as input parameters of some of the subroutines, together with the if-conditions that indicate whether a code block should be executed during the precomputation step or during the online computation step.

Figure 7 shows how the optimized program may be invoked by the interrupt service routine. Unlike what is shown in Figure 3, here, precomputation is performed when `msg` is not available but ambient energy is available (Line 4). When `msg` is available, it depends on whether ambient energy is still available. If ambient energy is not available, then online computation is performed (Line 8). However, if ambient energy is available, operations that access non-volatile memory will be skipped, which makes `wots_trans()` behaves exactly the same as the original program (Line 10).

3 Overview of Our Method

In this section, we first present the top-level procedure of our proposed method, and then outline the main technical challenges. In the next three sections, we will present our solutions to overcome these challenges.

3.1 The Top-Level Procedure

Algorithm 1 shows our top-level procedure. The input consists of the original program (P), the *online* input (OI) of the program, and the system constraint (C). The output is the optimized program (P').

Algorithm 1 The top-level procedure of our method.

```
   input  : original program $P$, online input $OI$, system constraint $C$
   output : optimized program $P'$
1  $PDG \leftarrow$ ConstructPDG ($P$);
2  $preSet \leftarrow$ IdentifyPreSet ($P$, $PDG$, $OI$);
3  $preSet^* \leftarrow$ OptimizePreSet ($preSet$, $PDG$, $C$);
4  $P' \leftarrow$ Transform ($P$, $PDG$, $preSet^*$);
5  return $P'$
```

For the running example in Figure 3, where the entry function is `wots()`, the online input is $OI = \{$`msg`$\}$, since `msg` is the only input value that must be ready at run time. C consists of a set of platform-dependent requirements, e.g., the size of non-volatile memory used to store coupons must be limited to $\leq$256 KB.

Inside Algorithm 1, our method first constructs a program dependency graph (PDG) for the program P. Then, our method uses the PDG and the set of variables in the *online* input OI to compute $preSet$, which is the set of instructions in P that may be precomputed. Next, it computes $preSet^*$, which is a subset of $preSet$ that represents the optimal solution to the constraint satisfiability problem. Finally, our method transforms the program P to a new program P' based on the information stored in both PDG and $preSet^*$.

Before presenting detailed algorithms in subroutines `IdentifyPreSet()`, `OptimizePreSet()` and `Transform()`, we would like to point out the main technical challenges.

3.2 The Technical Challenges

The first challenge, related to the subroutine `IdentifyPreSet()`, is the complex nature of the program dependency analysis. In Figures 3 and 4, for example, we observe that the subroutine `sign()` depends on `gen_key()`; furthermore, the subroutine `gen_sig()` invoked by `sign()` depends on `gen_key()`. It means that we must consider not only dependencies of instructions within each subroutine, but also dependencies between subroutines.

Moreover, since we aim to transform individual functions of the original program without changing the overall functional call structure, each function must be analyzed in all of its calling contexts, to figure out how the function body should be optimized. In Figure 4, for example, it means that since `gen_random()` is called by both `gen_key()` and `sign()`, we must consider both calling contexts.

The second challenge, related to the subroutine `OptimizePreSet()`, is optimizing the precomputation policy while satisfying a variety of system constraints. Given $preSet$ (which is the set of instructions that may be computed), we need to identify a proper subset. For the

MSP430 family of microcontroller units, a limiting factor may be the capacity of non-volatile memory, only part of which may be dedicated to coupon storage. In general, this is a non-linear optimization problem, e.g., the storage cost may not increase linearly, or even monotonically, as more instructions are added to the precomputation set.

In Figure 4, for example, the cost of precomputing only Lines 2-4 is $size$(`priv_key`) + $size$(`keyHash`), where $size$() denotes the size of non-volatile memory for storing the value. However, the cost of precomputing Lines 2-5 is $size$(`priv_key`) + $size$(`pub_key`), because `keyHash` no longer needs to be stored in non-volatile memory. Since $size$(`pub_key`) is much smaller than $size$(`keyHash`) in the W-OTS example, this means that precomputing one more line actually decreases the overall storage cost.

The third challenge, related to the subroutine `Transform()`, is the difficulty in preserving functional equivalence while allowing the program to change its execution order and data flow. For example, if we want to precompute Line 2 and Line 8 in Figure 4, we must modify the program to ensure that the original execution order (Line l_3 executed before Line l_8) changes to the new execution order (l_8 executed before l_3); at the same time, we must ensure that the original data flow `priv_key`$(l_2) - l_3, l_4, l_5 - l_8$ changes to `priv_key`$(l_2) - l_8 - l_3, l_4, l_5$. While doing so for this particular example may seem easy, in general, guaranteeing functional equivalence during such program transformation can be challenging.

4 Identifying the Precomputation Set

In this section, we present our method for computing $preSet$, as shown in Algorithm 2. It takes the original program P, the program dependency graph PDG, and the *online* input OI as parameters, and return $preSet$ as output.

Algorithm 2 The subroutine `IdentifyPreSet` (P, PDG, OI).

```
1 Let $pred(inst)$ be a predecessor node of instruction $inst$ in the $PDG$
2 $preSet \leftarrow$ {elementary instructions in $P$} $\cup$ ({input parameters of $P \setminus OI$)
3 while $\exists inst \in preSet$ and $\exists pred(inst) \notin preSet$ do
4     remove $inst$ from $preSet$
5 end
6 return $preSet$
```

Recall that $preSet$ is the set of instructions in P that may be computed ahead of time. Internally, our method computes $preSet$ in two steps. The first step is identifying the inter-procedural dependencies related to the online input OI. These dependencies will be captured by function such as $pred(inst)$, $preds(inst)$, and $succs(inst)$, which returns the predecessor, set of predecessors, and set of successors of an instruction $inst$, respectively. The second step is leveraging these dependencies to compute the instructions in $preSet$.

Inside Algorithm 2, initially, $preSet$ consists of all the *elementary instructions* and input parameters of P, except for the ones in OI. Variables in OI are excluded because they are the *online* variables. Here, an *elementary instruction* means that, during our analysis the instruction will be treated as a whole. First, non-function-call instructions are elementary instructions. Second, when an instruction invokes a function call, whether it is elementary depends on how many times the function is called. If the function is called only once, it is not treated as an elementary instruction; instead, we enter the function body to try to identify more precomputation opportunities. But if the function is called from multiple sites, we treat each call as an elementary instruction, meaning that we do not enter the function body

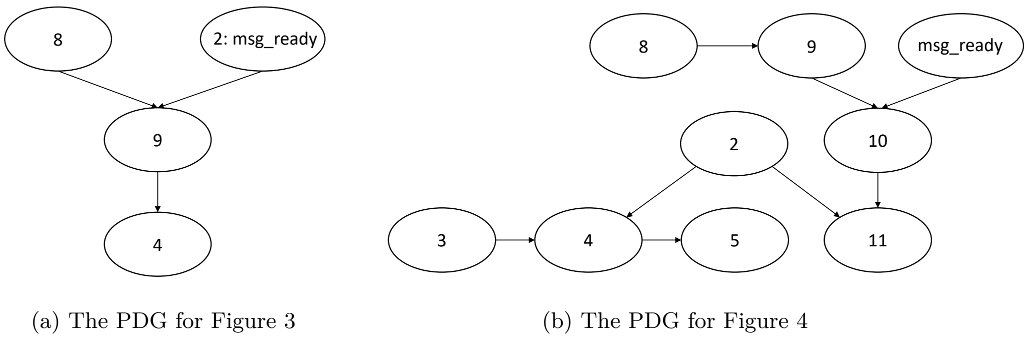


Figure 8 The program dependency graphs (PDGs) of the example W-OTS program. Here, each node represents an instruction, and the number is the instruction's line number in the program.

to explore further. This is a reasonable compromise since, when a function is called from multiple sites, the function body often implements some basic computation, e.g., generating a random number, and there is no need to split it further into the precomputation and online computation parts.

4.1 Inter-Procedural Dependencies

To identify the maximum set of instructions in $PreSet$ using Algorithm 2, we need the dependencies associated with the online input OI. These dependencies are more complex than what are typically available in the compiler. For example, by default, LLVM provides the control- and data-dependencies between instructions only within each function. However, we need to know dependencies not only within each function, but also between functions.

To identify inter-procedural dependencies, we first compute a PDG for each function, together with a call graph that represents the *caller-callee* relations of all functions in the program. We also extend LLVM to add the ability to determine whether a function call may change the content of a function parameter passed by reference or the value of a global variable. This is accomplished by traversing paths in the call graph and analyzing all of the functions involved in the path.

Next, we analyze the inter-procedural dependencies in a bottom-up fashion, according to the function call graph. Consider the example of the following two functions: `fun1(arg`$_1$`)` and `fun2(arg`$_2$`,arg`$_3$`)`, where the input parameter `arg`$_1$ of `fun1()` depends on the output parameter `arg`$_2$ of `fun2()`. Assume that `arg`$_3$ is also an output parameter of `fun2()`.

Assume that, inside the body of function `fun2()`, there is an instruction I that computes the value of `arg`$_2$. Furthermore, inside the body of function `fun1()`, there is another instruction I' that computes the value of `arg`$_1$.

While all of the intra-procedural dependencies may be computed in isolation, we must combine them to identify the inter-procedural dependencies, such as the dependency between I' of `fun1()` and I of `fun2()`.

Figure 8 shows a more concrete example, where the PDGs are constructed for the code snippets in Figures 3 and 4. Consider the edge $2 \rightarrow 11$ in Figure 8 (b), which represents the dependency between the instruction at Line 2 and the instruction at Line 11 of the program in Figure 4. It means the input parameter `priv_key` used by `sign()` at Line 11 comes from the output parameter `priv_key` of `gen_key()` at Line 2.

With the inter-procedure dependencies, we can define the notion of a *predecessor*, denoted by $pred()$. For example, in Figure 8 (b), due to the edge $2 \rightarrow 11$, we say that the instruction at Line 2 is a predecessor of the instruction at Line 11 inside the program shown by Figure 4.

4.2 Iteratively Computing *preSet*

Using the notion of a *predecessor* of an instruction $inst$, denoted $pred(inst)$, our method computes the $preSet$ according to the while-loop in Algorithm 2.

It starts with all elementary instructions and input parameters that are not in OI. Then, it removes any instruction ($inst$) whose predecessor $pred(inst)$ is not in $preSet$. There are two possible reasons why $pred(inst)$ is not in $preSet$: either it is in OI, or during the previous iteration, it has been removed from $preSet$. Thus, it is a fixed-point computation.

The correctness of the fixed-point computation can be understood as follows: By definition, the instruction $inst$ depends on its predecessor $pred(inst)$. If $pred(inst) \notin preSet$, meaning the predecessor instruction cannot be computed ahead of time, then the instruction $inst$ itself cannot be computed ahead of time either.

As an example, consider the instructions of W-OTS in Figure 4. For ease of presentation, we use l_i to represent the instruction at Line i, and we treat all instructions in this program as elementary instructions. Initially, we have $preSet = \{l_2 - l_5, l_8 - l_{11}\}$.

Next, we check if any of these instructions should be removed, based on the *predecessor* relation shown in Figure 8. The instruction l_{10} should be removed, since its predecessor (`msg_ready`) is not in $preSet$. Thus, we remove l_{10} from $preSet$.

The removal of l_{10} leads to the removal of l_{11} during the next iteration, since l_{10} is the predecessor of l_{10}. If l_{11} cannot be computed ahead of time, then l_{10} cannot be computed ahead of time either.

Thus, in the end, we have $preSet = \{l_2 - l_5, l_8 - l_9\}$.

▶ **Theorem 1.** *Our method for computing preSet is sound in that, for all inst* $\in$ *preSet, there is guarantee that the instruction (inst) can indeed be computed ahead of time.*

Proof: An instruction $inst$ remains in $preSet$ only if all of its predecessors are also in $preSet$. As long as the inter-procedural dependencies represented by the PDGs are an over-approximation of the actual dependencies, the $preSet$ is guaranteed to be an under-approximation of the set of instructions that may be computed ahead of time. It is an under-approximation because $pred(inst)$ is an over-approximation of the actual predecessors. Whenever $pred(inst) \notin preSet$, Algorithm 2 removes $inst$ from $preSet$.

The reason why $pred(inst)$ is an over-approximation is due to the nature of PDG-based analysis techniques. Refer to Horwitz et al. [23] and Reps et al. [36] for more information. ◀

4.2.1 Handling Loops

Similar to all other PDG-based analysis techniques [23, 36], our method has no problem in handling software code with loops. In most of the practical cases, computing the *predecessor* is straightforward. For example, the function call `sign()` at Line 9 in Figure 3 requires `msg` and `priv_key` to be available. These dependencies are due to data flow represented by the *definition-use* correspondence.

However, there are cases where definitions and uses do not have one-to-one mapping. For example, in Figure 9, the variable `i` used at Line 7 may be defined at either Line 2 or Line 5. In the context of data-flow analysis, the definition at Line 5 does not *kill* the definition at Line 2. Therefore, it may or may not be necessary to precompute Line 3-6 in order to precompute Line 7, for example, if `CNT[len-1]!=0xff`.

```
void increment_CNT(BYTE *CNT, int len){
    int i = len;
    while ((i > 0) && (CNT[i - 1] == 0xff)){
        CNT[i - 1] = 0;
        i--;
    }
    if (i) {
        CNT[i - 1]++;
    }
}
```

Figure 9 Code snippet taken from the benchmark program named `AES-CTR`.

Since our method is designed to be sound, to ensure that the optimized program is correct for all input values, it is allowed to over-approximate the predecessor relation, and then conservatively assume that an instruction can be precomputed *only if* all of its predecessors can be precomputed.

5 Optimizing the Precomputation Set

While all instructions in $preSet$ have been identified at this moment, it may not be beneficial to compute all of them ahead of time. In this section, we present our method for computing an optimal subset $preSet^* \subseteq preSet$. This is implemented in `OptimizePreSet`$(preSet, PDG, C)$, where C is the system constraint. Besides the characteristics of the hardware platform, such as the size of non-volatile memory, it also includes the characteristics of the software program, such as how often the encrypted sensor data must be transmitted to the remote server.

5.1 The Motivation

We use an example to illustrate the complex nature of the optimization problem, which in turn motivates our development of the constraint based solution.

Consider the W-OTS program in Figure 4 and its PDGs in Figure 8 (b). According to Algorithm 2, $preSet = \{l_2 - l_5, l_8 - l_9\}$. Since these instructions do not depend on the *online* input `msg`, in theory, they may be precomputed *as many times as possible*. However, due to the storage capacity, in practice, the number would have to be bounded.

Let $\mathcal{S}_i$ be a subset of $preSet$, called a precomputation choice, and m_i be the maximum number of times that $\mathcal{S}_i$ may be precomputed. Since each time $\mathcal{S}_i$ produces an intermediate result, or *coupon*, we also call m_i the *coupon count* (number of copies of this particular coupon). Let $\mathsf{NVM}(\mathcal{S}_i)$ be the storage cost for this coupon, and maxNVM be the storage capacity of the entire device. One precomputation choice for the running example is represented by $\mathcal{S}_1 = \{l_2\}$, where $m_1 \leq \mathsf{maxNVM}/\mathsf{NVM}(\mathcal{S}_1)$. That is, the coupon count m_1 is bounded only by the storage capacity.

Below are some other precomputation choices:

$$\begin{aligned}
&\mathcal{S}_2 = \{l_2 - l_5, l_8\}, \text{where } m_2 \leq \mathsf{maxNVM}/\mathsf{NVM}(\mathcal{S}_2)\\
&\mathcal{S}_3 = \{l_2 - l_5, l_8 - l_9\}, \text{where } m_3 \leq \mathsf{maxNVM}/\mathsf{NVM}(\mathcal{S}_3)\\
&\ldots
\end{aligned}$$

Let $n = |preSet|$, the total number of possible precomputation choices is $\Sigma_{i=1}^{n}\binom{n}{i}$. Since it leads to a combinatorial explosion, we cannot afford to enumerate them to decide which one is optimal.

The number of possible precomputation choices can be even higher than $\Sigma_{i=1}^{n}\binom{n}{i}$. For example, when $\mathcal{S}_{4a} = \{l_2 - l_5\}$ and $\mathcal{S}_{4b} = \{l_2 - l_5, l_8 - l_9\}$, if we allow the coupon counts m_{4a} and m_{4b} to have different values, they would be bounded only by the constraint $m_{4a} \times \mathsf{NVM}(\mathcal{S}_{4a}) + m_{4b} \times \mathsf{NVM}(\mathcal{S}_{4b}) \leq \mathsf{maxNVM}$. This leads to another combinatorial explosion.

While making a precomputation choice, we cannot consider instructions in isolation, since they may be dependent of each other. For example, precomputing one instruction may require precomputing another instruction. Recall that in the example program shown in Figure 4, we cannot precompute l_5 without precomputing l_4, because there is dependency from l_4 to l_5. In other words, $l_4 = pred(l_5)$.

All these challenges motivate us to define the constraint satisfiability problem, which allows us to consider all of the selected instructions as a whole, together with a variety of system constraints. Specifically, it allows us to consider the coupon count (m_i) and the coupon size $\mathsf{NVM}(\mathcal{S}_i)$ for each subset $\mathcal{S}_i \subseteq preSet$, together with system constraints such as the capacity of non-volatile memory used to store coupons computed by different instructions, and the inter-procedural dependencies between these chosen instructions.

5.2 The Problem Statement

Our goal is to compute the optimal subset, denoted $\mathcal{S}^* \subseteq preSet$, that satisfies the system constraint. For ease of presentation, assume that $\mathcal{S}$ represents a precomputation choice, while $V(\mathcal{S})$ represents the value (or benefit) of precomputing $\mathcal{S}$, and $C(\mathcal{S})$ represents the cost of precomputing $\mathcal{S}$. The optimization problem can be defined formally as follows:

$$\mathcal{S}^* = \underset{\mathcal{S}\subseteq preSet}{\mathsf{argmax}}\, V(\mathcal{S}) \quad \text{subject to} \quad C(\mathcal{S}) \leq \mathsf{maxNVM} \tag{5.2}$$

In other words, the optimal subset is the subset $\mathcal{S}$ that maximize the value $V(\mathcal{S})$ while keeping to cost $C(\mathcal{S})$ under control. Based on our analysis above, explicitly enumerating the possible solutions would lead to a combinatorial explosion. Thus, we encode the problem as a set of logical constraints and then solve these constraints using an off-the-shelf SMT solver.

One advantage of the *constraint-based approach* is the flexibility in modeling various tradeoffs. While it is easy to compute the coupon size or the coupon count individually, finding the right combination may be due to the fact that they are inter-dependent.

Another advantage of our approach is the flexibility in modeling the chain of influence; that is, precomputing one instruction (e.g., l_4 of `gen_key` in Figure 4) may require precomputing another instruction (e.g., l_3).

Yet another advantage is the ability to bound the total cost of storing coupons from different instructions. As mentioned earlier, precomputing more instructions may not always increase the storage cost. In Figure 4, if we precompute $l_3 - l_4$ but not l_5, we need to store both `pub_key` and `keyHash`, the latter of which is an array of 108 bytes; but if we precompute $l_3 - l_5$, we only need to store `pub_key`, which is an array of 32 bytes.

5.3 Defining the Value and Cost Functions

First, we define the energy saving (*value*) and storage overhead (*cost*) of precomputing a set of instructions.

5.3.1 Value

Since the value of precomputing one instruction may depend on which other instructions are precomputed, we can only define it based on which other instructions are chosen. Since an instruction *inst* may be precomputed only if all its *predecessors* are precomputed, we define the value of precomputing *inst* based on the predecessor relation.

Let $\mathcal{S}$ be the set of chosen instructions, and $v(inst \mid \mathcal{S})$ be the *value* of precomputing *inst* in the presence of $\mathcal{S}$. We have

$$v(inst \mid \mathcal{S}) = \begin{cases} E(inst) & \text{if } preds(inst) \subseteq \mathcal{S} \\ -\infty & \text{otherwise} \end{cases}$$

Here, $E(inst)$ is the energy saved by precomputing *inst*, and $preds(inst)$ is the set of all predecessors of *inst* in the PDG. We use the large value $-\infty$ to avoid precomputing *inst* before all of its predecessors in $preds(inst)$ are precomputed.

With the values of precomputing individual instructions, we define the *value* of precomputing the entire set $\mathcal{S}$ as follows:

$$V(\mathcal{S}) = \sum_{inst \in \mathcal{S}} v(inst \mid \mathcal{S}).$$

For the running example in Figure 4 and Figure 8 (b), we have $V(\{l_2\}) = En(l_2)$. We also have $V(\{l_2, l_5\}) = -\infty$ since l_5 cannot be selected when its predecessors $l_3 - l_4$ are not selected.

5.3.2 Cost

Unlike the value $v(inst)$, which depends only on the predecessors of *inst*, the cost of precomputing *inst* depends also on its *successors* in the PDG.

Let $\mathcal{S}$ be the set of chosen instructions, and $c(inst \mid \mathcal{S})$ be the *cost* of precomputing *inst* in the presence of $\mathcal{S}$. In Figure 4, for instance, we have

$$c(l_3 \mid \mathcal{S}) = \begin{cases} 0 & \text{if } l_4, l_5 \in \mathcal{S} \\ \mathsf{NVM}(\texttt{keyHash}) & \text{otherwise} \end{cases}$$

and

$$c(l_4 \mid \mathcal{S}) = \begin{cases} 0 & \text{if } l_2, l_3 \in \mathcal{S} \\ +\infty & \text{otherwise} \end{cases}$$

That is, if $l_3 - l_5$ are selected, we do not need to store `keyHash`; but if $l_4 - l_5$ are not selected, we need to store `keyHash`. Thus, the cost of precomputing l_3 depends on if $(l_4 - l_5)$ are selected. Here, the large value $+\infty$ is used to avoid selecting instructions whose predecessors in the PDG are not selected.

With the costs of precomputing individual instructions, we define the cost of precomputing the entire set $\mathcal{S}$ as follows:

$$C(\mathcal{S}) = \sum_{inst \in \mathcal{S}} c(inst \mid \mathcal{S}).$$

5.4 Symbolic Encoding of the Constraints

We construct an SMT formula $\Psi = \Phi_{Dep} \wedge \Phi_{Value} \wedge \Phi_{Cost}$, where the subformula Φ_{Dep} captures the dependencies that we have computed in the previous section, Φ_{Value} captures the value constraint, and Φ_{Cost} captures the cost constraint. Thus, a satisfying assignment to Ψ corresponds to $\mathcal{S}^* \subseteq preSet$.

5.4.1 Dependency Constraint

Φ_{Dep} encodes the dependency relations captured by edges of the inter-procedural PDG. Specifically, for each dependency edge (n_1, n_2), we add a Boolean constraint $(\neg n_2 \vee n_1)$, where n_1 and n_2 are Boolean variables indicating whether these nodes are precomputed, and the constraint means that, if n_2 is true, then n_1 must also be true. Therefore, n_2 being precomputed implies that n_1 is also precomputed. Then, all these individual constraints are conjoined to form Φ_{Dep}. As an example, consider the PDG in Figure 8 (b): the dependency constraints include $(\neg l_4 \vee l_3) \wedge (\neg l_4 \vee l_2) \wedge (\neg l_5 \vee l_4) \wedge (\neg l_9 \vee l_8)$.

5.4.2 Value Constraint

Φ_{Value} encodes the value of precomputing each instruction. Since Φ_{Dep} already guarantees that an instruction is precomputed only if all its predecessors (as in the PDG) are precomputed, the encoding becomes straightforward. That is, if *inst* is selected, then $v(inst) = E(inst)$; otherwise $v(inst) = 0$. The total value of precomputing the set of instructions in *preSet* is simply the sum of all the individual values. In Figure 4, the value of precomputing each instruction l_i, where $i = 2, 3, \ldots, 5, 8, 9$, would be $v(l_i) = (l_i\ ?\ E(l_i) : 0)$ and the total would be $V(\mathcal{S}) = \sum v(l_i)$.

5.4.3 Cost Constraint

Φ_{Cost} encodes the cost of precomputing the chosen instructions. Recall that the cost of precomputing *inst* depends on not only if its predecessors are precomputed but also if its successors are precomputed. Since Φ_{Dep} guarantees to select the predecessors whenever *inst* is selected, here we only need to deal with the set of successors, denoted $succs(inst)$.

In general, precomputing *inst* increases storage cost only when its result (coupon) is used by some of the successors in the online computation step; otherwise, there is no need to save the coupon. For example, the cost of precomputing l_3 in Figure 4 is zero if instructions in $succs(l_3)$ =$\{l_4,\, l_5\}$ are also precomputed.

For the entire program shown in Figure 4, the cost constraint would be

$$
\begin{aligned}
&(\ c(l_2) = (\neg l_2 \vee l_3 \wedge l_4 \wedge l_5 \wedge l_{send})\ ?\ 0 : \mathsf{NVM}[\texttt{priv\_key}]\ ) \wedge \\
&(\ c(l_3) = (\neg l_3 \vee l_4)\ ?\ 0 : \mathsf{NVM}[\texttt{keyHash}]\ ) \wedge \\
&(\ c(l_4) = (\neg l_4 \vee l_5)\ ?\ 0 : \mathsf{NVM}[\texttt{keyHash}]\ ) \wedge \\
&(\ c(l_5) = \neg l_5\ ?\ 0 : \mathsf{NVM}[\texttt{pub\_key}]\ ) \wedge \\
&(\ c(l_8) = (\neg l_8 \vee l_9)\ ?\ 0 : \mathsf{NVM}[\texttt{rand}]\ ) \wedge \\
&(\ c(l_9) = \neg l_9\ ?\ 0 : \mathsf{NVM}[\texttt{sig}]\ ) \\
&(\ C(\mathcal{S}) = c(l_2) + c(l_3) + c(l_4) + c(l_5) + c(l_8) + c(l_9)\ ) \wedge \\
&(\ C(\mathcal{S}) \leq \mathsf{maxNVM}\ )
\end{aligned}
$$

5.5 Solving the Constraints

After constructing the entire SMT formula Ψ, we solve it using the Z3 SMT solver [16]. Specifically, we use Z3's `optimize` interface iteratively to search for the optimal solution. This is done by insisting that the total value $V(\mathcal{S})$ shown in Equation (5.2) is greater than a given constant value; then, we find the maximum constant by gradually increasing the value of the constant as long as Z3 can still find a satisfying solution.

6 Transforming the Program

We now explain the subroutine $\texttt{Transform}(P, PDG, preSet^*)$, which transforms the original program P to a new program P' to implement $preSet^*$. Recall that in Figure 6, we gave an example of such a transformed program for W-OTS. There are two important properties of the program P': (1) it retains the overall function call structure in P and (2) it changes the body of each function to implement both the precomputation and online computation steps.

6.1 The Terminology

For each function f in the program P, we must separate the precomputation instructions from the online computation instructions. This leads to a partition of the program to segments, $\{S_1, \tilde{S_2}, S_3, \tilde{S_4}, ...\}$, where S_i represents a precomputation segment and $\tilde{S_j}$ represents an online computation segment. A *segment* is a maximal set of instructions that may execute continuously during precomputation or online computation.

Consider an example program $P = \{S_1, \tilde{S_2}, S_3, \tilde{S_4}\}$ whose original execution order is $S_1 \rightarrow \tilde{S_2} \rightarrow S_3 \rightarrow \tilde{S_4}$. In the transformed program P', however, the execution order must be changed to $S_1 \rightarrow S_3 \rightarrow \tilde{S_2} \rightarrow \tilde{S_4}$. In general, changes in the execution order lead to changes in the data flow.

Before discussing changes in the data flow, we define the terminology.

- Let $def(x)$ be an instruction that defines the value of variable x, and $use(x)$ be an instruction that uses the value. The two instructions may form a *def-use* pair.
- Given two segments S_i and $\tilde{S_j}$, where $def(x) \in S_i$ and $use(x) \in \tilde{S_j}$, we represent the data-flow edge (or def-use pair) as $\langle S_i, \tilde{S_j} \rangle (x)$.
- Let $Val[x, S_i]$ denote the value of x at the end of executing the segment S_i.
- A variable x is *live* at a program location p if its value is used before it is defined again along *some* path from p to the exit.

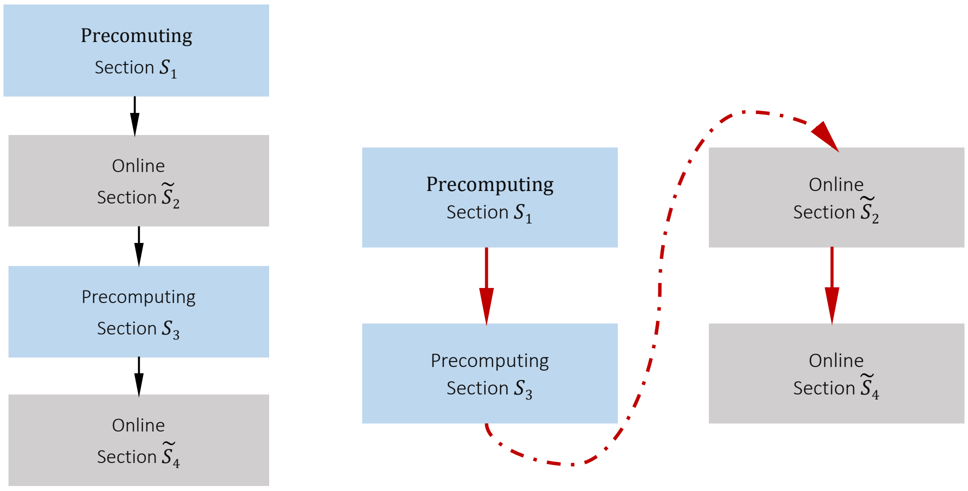
6.2 The Problem

Now, we show an example where changes in the execution order bring unexpected changes of the data flow.

▶ **Example 6.1.** *In program $P = \{S_1, \tilde{S_2}, S_3, \tilde{S_4}\}$, assume that $def_1(x) \in S_1$, $def_2(x) \in \tilde{S_2}$, $use(x) \in \tilde{S_4}$. Due to the execution order, the def-use chain contains only $def_2(x)$ and $use(x)$, meaning the value of x used in $\tilde{S_4}$ should be from $def_2(x)$.*

In the original execution order $S_1 \rightarrow \tilde{S_2} \rightarrow S_3 \rightarrow \tilde{S_4}$, the value $Val[x, S_3]$ comes from $def_2(x)$, and the variable x is live in S_3, since $Val[x, S3]$ will be used in $\tilde{S_4}$.

In the new program, however, since the execution order is changed to $S_1 \rightarrow S_3 \rightarrow \tilde{S_2} \rightarrow \tilde{S_4}$, without our intervention, the value $Val[x, S3]$ would come from $def_1(x)$, and the variable x would *no longer* be live in S_3. Such unexpected changes of the data flow would change the semantics of the program. This is illustrated by Figure 10.


Precomuting Section S_1
Online Section $\tilde{S}_2$
Precomputing Section S_3
Online Section $\tilde{S}_4$
Precomputing Section S_1
Precomputing Section S_3
Online Section $\tilde{S}_2$
Online Section $\tilde{S}_4$

(a) Execution order of P (b) Execution order of new program P'

Figure 10 Difference in execution order means P and P' are no longer functionally equivalent.

In general, it can be challenging to preserve the data flow while allowing the change of execution order. While the technique of *checkpointing* has been used in intermittent computing systems [28, 42, 31], it cannot solve our problem because checkpointing does not involve splitting a program into two parts and then executing the two parts in an interleaved order. For the program in Example 6.1, specifically, checkpointing techniques would have failed to preserve the data flow.

To understand why checkpointing would fail, consider the fact that variable x is *live* at the end of $\tilde{S}_2$, at the end of S_3, and at the start of $\tilde{S}_4$. Checkpointing would insert `nvm_ST`$(Val[x, \tilde{S}_2])$ at the end of $\tilde{S}_2$ and insert `nvm_ST`$(Val[x, S_3])$ at the end of S_3. It would also insert `nvm_LD`$(Val[x, \tilde{S}_2])$ and `nvm_LD`$(Val[x, S_3])$ at the start of $\tilde{S}_4$.

When executing P' ($S_1 \to S_3 \to \tilde{S}_2 \to \tilde{S}_4$), `nvm_LD`$(Val[x, S_3])$ would over-write `nvm_LD`$(Val[x, \tilde{S}_2])$; thus, the value of x used in $\tilde{S}_4$ would be $Val(x, S_3) = def_1(x)$. However, in the original program, the value of x used in $\tilde{S}_4$ is $def_2(x)$.

The fundamental reason why *checkpointing* techniques are ill-suited for our project is that the *liveness* property of a program variable, which forms the theoretical foundation of the checkpointing techniques, is not preserved by the split of the original program into the precomputation and online computation parts. Thus, instead of relying on the *liveness* property, our method relies on the *def-use* relations.

6.3 The Baseline Method

We first present the baseline method using the *def-use* relations, and then present the optimized method in the next subsection.

Since we treat each segment as an atomic unit during transformation, we only need to consider the *def-use* relations between segments. Thus, whenever two segments have *def-use* relations, there can only be three scenarios:

- (I) $\langle S_i, S_j \rangle$, meaning both are precomputation segments;
- (II) $\langle S_i, \tilde{S}_j \rangle$, meaning S_i is a precomption and $\tilde{S}_j$ is an online computation; and
- (III) $\langle \tilde{S}_i, \tilde{S}_j \rangle$, meaning both are online computation segments.

The fourth scenario, $\langle \tilde{S_i}, S_j \rangle$, is impossible due to our method for computing *preSet*.

In other words, a *use* in a precomputation segment always comes from a *definition* in a precomputation segment, whereas a *use* in an online computation segment may come from a definition in a precomputation or an online computation segment.

Furthermore, it suffices to handle only type (II) case $\langle S_i, \tilde{S_j} \rangle$, because for the other two cases, the value *can be* propagated directly between the two segments of the same type.

To maintain the *def-use* chains between precomputation and online computation segments in the type (II) case, we must insert `nvm_LD` and `nvm_ST` instructions at the proper *def* and *use* locations.

Thus, our baseline method can be summarized as follows: For each data-flow edge $\langle S_i, \tilde{S_j} \rangle(x)$, we insert `nvm_ST`$(Val[x, S_i])$ at the end of S_i, and insert `nvm_LD`$(Val[x, S_i])$ at the start of $\tilde{S_j}$.

Recall the scenario shown in Example 6.1, where the *def-use* chain contains only $def_2(x)$ and $use(x)$. According to our baseline method, no NVM operation needs to be added, since the *def-use* is of the type (III). The value of x used in $\tilde{S_4}$ comes directly from $def_2(x)$.

6.4 The Optimized Method

Now, we present an optimization to avoid *redundant* NVM operations inserted by the baseline method. To understand why some of the NVM operations inserted by our baseline method may be redundant, consider the following example.

▶ **Example 6.2.** *In $\{S_1, \tilde{S_2}, S_3, \tilde{S_4}\}$, assume that $def(x) \in S_1$, $use_1(x) \in \tilde{S_2}$, $use_2(x) \in \tilde{S_4}$, and the def-use chain contains both $def(x)$–$use_1(x)$ and $def(x)$–$use_2(x)$. Our baseline method would insert*

- `nvm_ST`$(Val[x, S_1])$ *after S_1 (twice);*
- `nvm_LD`$(Val[x, S_1])$ *before $\tilde{S_2}$;*
- `nvm_LD`$(Val[x, S_1])$ *before $\tilde{S_4}$.*

However, executing `nvm_LD`$(Val[x, S_1])$ *before $\tilde{S_4}$ is redundant because the value of x can be propagated directly from $\tilde{S_2}$.*

To avoid the redundant operations, we should insert `nvm_LD` of a $def(x)$ at the start of the earliest online computation segment where $def(x)$ is available. For the program in Example 6.2, the earliest segment is $\tilde{S_2}$, which means we should insert `nvm_LD`$(Val[x, S_1])$ right before $\tilde{S_2}$.

Thus, our optimized method can be summarized as follows: For each data-flow edge $\langle S_i, \tilde{S_j} \rangle(x)$ that we have not inserted `nvm_ST`$(Val[x, S_i])$ after S_i, insert `nvm_ST`$(Val[x, S_i])$ after S_i and insert `nvm_LD`$(Val[x, S_i])$ before $\tilde{S}_{i+1}$.

To understand the benefit of this optimization, let us compare the data flows of the following two programs. If, for example, in the original program, $Val[x, S_i]$ is available (and not killed) in the range

$$end[S_i] \rightarrow \tilde{S}_{i+1} \rightarrow S_{i+2} \rightarrow \tilde{S}_{i+3} \rightarrow \cdots \tag{1}$$

and in the transformed program, $Val[x, S_i]$ is available (and not killed) in the range

$$end[S_i] \rightarrow S_{i+2} \rightarrow S_{i+4} \rightarrow S_{i+6} \rightarrow \cdots \tag{2}$$

and `nvm_LD` $Val[x, S_i]$ has been inserted before $\tilde{S}_{i+1}$ in the transformed program, the loaded value will also be available in the entire range

$$\tilde{S}_{i+1} \rightarrow \tilde{S}_{i+3} \rightarrow \tilde{S}_{i+5} \rightarrow \tilde{S}_{i+7} \rightarrow \cdots \tag{3}$$

Therefore, we can avoid the other (redundant) `nvm_LD` operations before $\tilde{S}_{i+3} \ldots \tilde{S}_{i+7}$.

6.5 The Transformation Algorithm

To sum up, our optimized method for transforming each function f of the original program based on *preSet** is presented in Algorithm 3.

Algorithm 3 Transforming a function f in program P based on $preSet^*$.

```
1 Partition $f$ into segments $\{S_i\}$ and segments $\{\tilde{S}_j\}$;
2 Add *if-condition* to each segment using precom_flag and online_flag;
3 foreach *data-flow edge denoted* $\langle S_i, \tilde{S}_j \rangle(x)$ do
4     if *there is no* nvm_ST($Val[x, S_i]$) *after segment* $S_i$ then
5         Add nvm_ST($Val[x, S_i]$) after $S_i$;
6         Add nvm_LD($Val[x, S_i]$) before $\tilde{S}_{i+1}$;
7     end
8 end
```

Our method first partitions the instructions in function f to precomputation segments $\{S_i\}$ and online computation segments $\{\tilde{S}_j\}$. Next, it inserts *if-condition* to each segment using the two flags, to differentiate the three use cases. Finally, for each data-flow edge $\langle S_i, \tilde{S}_j \rangle(x)$, it insert NVM operations to store the value of variable x computed in S_i (the coupon) at the end of segment S_i.

While in the baseline method, the coupon is loaded from NVM at the start of $\tilde{S}_j$, in the optimized method, the coupon is loaded at the start of the online computation segment $\tilde{S}_{i+1}$. By loading the coupon earlier, we have the opportunity to eliminate many redundant NVM operations.

7 Experiments

We have implemented our method in a software tool, named CouponMaker, which builds upon the LLVM compiler platform [27] and the Z3 SMT solver [16]. We leverage LLVM to parse the C code of the original program, conduct inter-procedural dependency analysis and implement the semantic-preserving transformation. We use Z3 to solve the constraint optimization subproblems. In total, our implementation adds 1,852 lines of C++ code.

Our tool generates the LLVM bit-code of the optimized program as output, which in turn is compiled to machine code for the MSP430 MCU. To evaluate the performance of the optimized program, we use the cycle-accurate emulator MSPSim [34]. Specifically, we use MSPSim to compute the latency and energy consumption of the optimized program, and compare them with the latency and energy consumption of the original program.

7.1 Benchmarks

We evaluated CouponMaker on 26 benchmark programs, which are C programs implementing lightweight cryptographic protocols. In total, they have 31,113 lines of C code. The statistics of these programs are shown in Table 1, where Columns 1-3 show the name, category, and source of each program, and Column 4 shows the number of lines of code (LoC).

The benchmark programs fall into two groups. The first group consists of programs that compute one-time signatures (W-OTS and Lamport) and the second group consists of programs that implement block-ciphers (e.g., AES and Camellia). For each of the eight

Table 1 Statics of the benchmark programs.

Name	Category	Source	LoC
W-OTS	One-time signature	Merkle signature [4]	1, 062
Lamport	One-time signature	Lamport signature[2]	339
AES	Encryption Algorithm	OpenSSL[5]	1,572
Blowfish	Encryption Algorithm	OpenSSL[3]	946
Camellia	Encryption Algorithm	OpenSSL[5]	708
CAST	Encryption Algorithm	OpenSSL[5]	1,084
DES	Encryption Algorithm	avr-crypto-lib[1]	1,277
GOST	Encryption Algorithm	OpenSSL[5]	357
skipjack	Encryption Algorithm	avr-crypto-lib[1]	475
SEED	Encryption Algorithm	OpenSSL[5]	1,277

block-cipher programs, we also configure it in three different modes, marked by suffixes -OFB, -CFB, and -CTR, respectively.

Our experiments were conducted on a computer with 2 GHz Intel Core i5 CPU and 16 GB memory. These experiments were designed to answer the following questions:

- Is CouponMaker efficient in optimizing the benchmark programs?
- Are the optimized programs better than the original programs in terms of both energy efficiency and latency?

7.2 Performance of the Optimization Tool

Table 2 shows the results of evaluating the optimization tool. Column 1 shows the benchmark name. Column 2 shows the total running time in seconds. Column 3 shows the size of $preSet$, which is the set of instructions that may be precomputed. Columns 4-5 compare the sizes of the original and optimized programs, where the sizes are measured in the number of bytes of the LLVM bit-code. Columns 6-8 show the details of the coupons stored in non-volatile memory, including the number of coupons, and the total bytes, and whether the coupons may be precomputed multiple times (copies).

Specifically, ∞ in the last column means the coupons may be precomputed as many times as possible, while 1 means they may be precomputed only once.

For programs that compute one-time signatures (W-OTS and Lamport), for example, a theoretically unbounded number of signatures (coupons) may be precomputed. For block-cipher programs in the -OFB mode, the ciphertext of the first block may also be precomputed as many times as possible (after the first block becomes available), and in the -CNT mode, the counter CNT may be incremented as many times as possible and then pre-encrypted for future use.

For block-cipher programs in the -CFB mode, however, precomputation can only be done once per block, i.e., after the current block arrives.

The results show that our proposed technique is able to analyze, optimize, and transform all benchmark programs quickly. The total running time is limited to a few seconds. Moreover, the size of the program before and after optimization does not change significantly. Furthermore, the number and size of precomputed coupons are significant for all programs.

7.3 Performance of the Optimized Programs

Table 3 shows the result of evaluating the performance of the optimized programs. These results were obtained using the MSPSim tool for MSP430FR599x [24]. Since MSPSim requires the programs to be executed under concrete test inputs, for one-time signature

Table 2 Performance of the analysis tool CouponMaker.

Name	Time (s)	PreSet Size	Program Size orig.	Program Size opti.	Coupon Size num	Coupon Size bytes	Coupon Size copies
W-OTS	5.26	1,632	16,116	21,704	3	1,152	∞
Lamport	4.08	1,000	14,268	19,116	2	512	∞
AES-OFB	3.35	3,964	52,636	57,984	1	16	∞
AES-CFB	3.62	3,964	56,162	56,168	1	16	1
AES-CTR	3.73	4,064	53,164	58,584	1	16	∞
Camellia-OFB	3.37	1,412	20,228	25,276	1	16	∞
Camellia-CFB	3.30	1,412	20,696	25,788	1	16	1
Camellia-CTR	3.89	1,460	24,964	29,984	1	16	∞
DES-OFB	3.11	2,072	26,384	26,496	1	8	∞
DES-CFB	3.14	2,072	26,432	26,644	1	8	1
DES-CTR	3.05	2,112	26,896	27,556	1	8	∞
Blowfish-OFB	3.38	1,196	16,200	21,308	1	8	∞
Blowfish-CFB	3.27	1,196	16,180	21,288	1	8	1
Blowfish-CTR	3.70	1,242	16,636	21,724	1	8	∞
skipjack-OFB	3.09	1,896	34,452	39,552	1	8	∞
skipjack-CFB	3.26	1,896	34,404	39,536	1	8	1
skipjack-CTR	3.32	1,940	34,864	40,008	1	8	∞
GOST-OFB	2.79	596	12,508	17,504	1	8	∞
GOST-CFB	3.16	596	12,492	17,484	1	8	1
GOST-CTR	3.01	844	12,952	17,984	1	8	∞
SEED-OFB	2.67	196	31,120	36,384	1	8	∞
SEED-CFB	2.68	196	31,100	36,368	1	8	1
SEED-CTR	3.11	340	31,564	36,852	1	8	∞
CAST128-OFB	2.49	352	46,628	51,748	1	8	∞
CAST128-CFB	2.74	352	46,608	51,732	1	8	1
CAST128-CTR	3.00	396	47,064	52,228	1	8	∞

programs (W-OTS and Lamport), we obtain the test inputs by signing a fixed-length message; for block-cipher programs, we obtain the test inputs by encrypting sensor data that represent a sequence of temperature measurements.

In the result table, Column 1 shows the benchmark name. Column 2 shows the energy (μJ) consumed by the original program. Columns 3-4 show the energy (μJ) consumed by the optimized program, which is divided into the precomputing and online steps. Recall that in energy-harvesting applications, energy reported in the $E(pre)$ column is considered to be free. Thus, the ratio in Column 5 represents the actual performance improvement.

The results show that the optimized programs significantly outperform the original programs in terms of energy efficiency. The improvement ranges from 2.14X to 29.32X. We also compared the latency of the original and optimized programs and observed a similar performance improvement. These results show that our precomputation based optimization method is effective in reducing the energy cost.

7.4 Impact of the Precomputation Policy

Finally, we compare the impact of the precomputation policy by computing the energy saving per unit use of non-volatile memory storage, measured by $qf = (En(ori) - En(on))/Size(coupon)$, where qf stands for quality factor. The results are shown in Figure 11, where the x-axis shows the benchmark programs and the y-axis shows the quality factors (qf) achieved by the baseline and optimized methods for program transformation (Section 6).

Here, *optimal* corresponds to the optimized precomputation policy ($preSet^*$), while

Table 3 Evaluating reduction in energy cost on MSP430.

Name	Original Program	Optimized Program		Improvement
	$E(ori)$	free $E(pre)$	$E(on)$	$E(ori)/E(on)$
W-OTS	115565.43	56114.99	49576.70	2.3X
Lamport	355.91	287.31	82.71	4.3X
AES-OFB	89.06	87.96	4.85	18.4X
AES-CFB	90.67	87.96	6.46	14.0X
AES-CTR	89.23	88.15	3.36	26.5X
Camellia-OFB	28.66	27.56	4.85	5.9X
Camellia-CFB	30.27	27.56	6.46	4.7X
Camellia-CTR	28.84	27.75	4.87	5.9X
DES-OFB	198.84	197.87	5.42	36.7X
DES-CFB	200.56	197.88	7.14	28.1X
DES-CTR	199.18	198.25	5.45	36.6X
Blowfish-OFB	15.63	14.66	5.43	2.9X
Blowfish-CFB	17.35	14.66	7.14	2.4X
Blowfish-CTR	15.97	12.64	4.01	4.0X
skipjack-OFB	26.16	25.20	5.42	4.8X
skipjack-CFB	29.33	25.20	8.58	3.4X
skipjack-CTR	26.73	26.06	5.71	4.7X
GOST-OFB	29.10	29.01	2.59	11.3X
GOST-CFB	29.83	29.01	3.32	9.0X
GOST-CTR	29.65	29.61	2.62	11.3X
SEED-OFB	20.32	19.21	4.85	4.2X
SEED-CFB	21.92	19.21	6.45	3.4X
SEED-CTR	20.49	17.64	3.22	6.4X
CAST128-OFB	164.89	161.86	16.89	9.8X
CAST128-CFB	170.24	161.86	22.24	7.7X
CAST128-CTR	165.95	163.05	16.99	9.8X

baseline corresponds to the initial precomputation policy ($preSet$), which means all instructions in $preSet$ are precomputed. Note that higher qf values correspond to better results. Overall, the optimized precomputation policy corresponds to significantly better results.

For W-OTS, qf(optimal) is also significantly higher than qf(baseline). However, the qf values for W-OTS are not included in the figure, to avoid making the rest of the bar chart less readable. This is because W-OTS executes several orders-of-magnitude more clock cycles than the other programs, and thus has much higher qf values.

8 Related Work

While prior work has shown the feasibility of optimizing energy-harvesting applications using precomputation [41], the optimization must be performed manually by domain experts. To the best of our knowledge, this is the first constraint based method for optimizing the software code automatically. Compared to the manual optimization of Suslowicz et al. [41], in particular, our method can automatically complete all of the optimization work with comparable performance. Moreover, our method can support additional constraints for optimization, which the manual method cannot deal with easily. Since our method aims to preserve the original program semantics, it is not meant for scenarios where the underlying algorithms are intended to be rewritten according to some mathematical rules [10, 11] – automation for such transformation is beyond the scope of this work.

Our method differs from intermittent computing techniques aimed to improve general-purpose systems with a strong and yet unstable power supply; these techniques [37, 31, 28, 42]

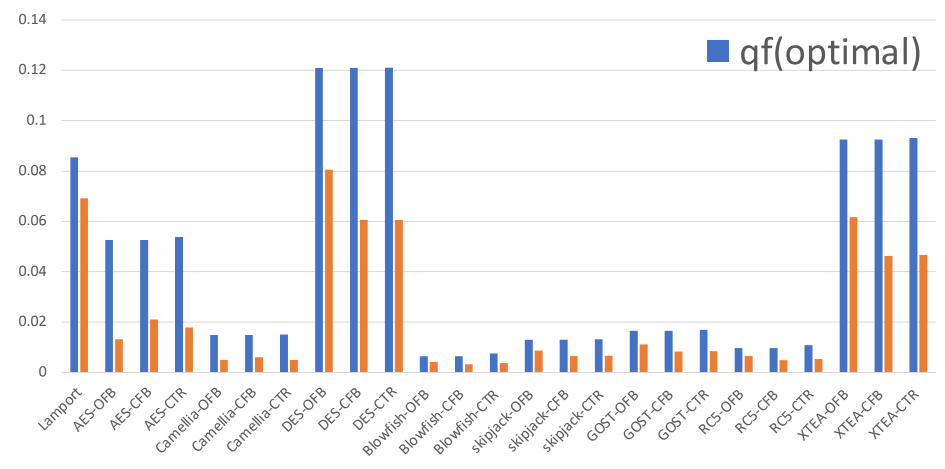


Figure 11 The impact of the precomputation policy on performance improvement. Here, baseline corresponds to *preSet* and optimal corresponds to $preSet^*$.

focus on recovering from power loss using *checkpointing*, avoiding the costly register accesses or reducing the cost for loop-heavy programs [20, 19]. There are also techniques for robustly supporting peripherals [40, 32]. However, none of them considers the scenario where ambient energy source is ample but the computing device is idle, let alone leveraging precomputation to reduce the energy cost.

There are also techniques for programming transiently powered computers with both volatile and non-volatile memory, for example, by leveraging the application's memory access patterns to manually optimize the data placement [14, 28, 30], or the mapping of code sections to either volatile or non-volatile memory [25] based on where the optimal energy consumption could be achieved. There are also efficient checkpointing techniques [22, 7] for CPUs with fully non-volatile main memory. However, none of them focuses on automated program optimization based on precomputation.

While our focus in this work is on optimizing software for energy-harvesting devices, the underlying ideas may be applied to other applications of similar nature, e.g., precomputation for Trusted Authority (TA) in the context of secure multi-party computation (e.g., multi-party learning and predicting[43, 18]). Since the application domain is significantly different, to deal with software used in such applications, our LLVM based implementation may need to be updated accordingly to handle certain language constructs – we leave this for future work.

9 Conclusion

We have presented a constraint-based method for optimizing the efficiency of software code running on devices powered by electricity harvested from the environment. Our method relies on static program analysis to identify instructions that may be precomputed, constraint solving to compute an optimal subset, and compiler transformation to generate the new software code. Our experimental evaluation on a large number of benchmark programs shows that the proposed method can handle all of the benchmark programs quickly, and the optimized programs significantly outperform the original programs in terms of both energy efficiency and latency.

References

1 The avr-crypto-lib software package. `https://github.com/cantora/avr-crypto-lib`. Accessed: 2019-09-26.

2 The lamport_signature software package. `https://github.com/detomastah/lamport_signature`. Accessed: 2019-09-26.

3 The Libgcrypt software package. `https://www.gnupg.org/software/libgcrypt/index.html`. Accessed: 2019-09-26.

4 OpenSSL. `https://www.openssl.org`. Accessed: 2019-09-26.

5 Shorter Merkle Signatures. `https://www.openssl.org`. Accessed: 2019-09-26.

6 The tiny Dutch startup solving the IoT industry's battery problem. `https://sifted.eu/articles/nowi-dutch-startup-solving-iot-battery-problem/`. Accessed: 2020-08-04.

7 Saad Ahmed, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Efficient intermittent computing with differential checkpointing. In Jian-Jia Chen and Aviral Shrivastava, editors, *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2019, Phoenix, AZ, USA, June 23-23, 2019*, pages 70–81. ACM, 2019.

8 Saad Ahmed, Muhammad Nawaz, Abu Bakar, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Demystifying energy consumption dynamics in transiently powered computers. *ACM Trans. Embed. Comput. Syst.*, 19(6):47:1–47:25, 2020.

9 James Allen, Matthew Forshaw, and Nigel Thomas. Towards an extensible and scalable energy harvesting wireless sensor network simulation framework. In Walter Binder, Vittorio Cortellessa, Anne Koziolek, Evgenia Smirni, and Meikel Poess, editors, *Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 39–42. ACM, 2017.

10 Giuseppe Ateniese, Giuseppe Bianchi, Angelo Capossele, and Chiara Petrioli. Low-cost standard signatures in wireless sensor networks: a case for reviving pre-computation techniques? In *Network and Distributed System Security Symposium*, 2013.

11 Giuseppe Ateniese, Giuseppe Bianchi, Angelo T Capossele, Chiara Petrioli, and Dora Spenza. Low-cost standard signatures for energy-harvesting wireless sensor networks. *ACM Transactions on Embedded Computing Systems*, 16(3):64, 2017.

12 Domenico Balsamo, Alex S Weddell, Geoff V Merrett, Bashir M Al-Hashimi, Davide Brunelli, and Luca Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7(1):15–18, 2015.

13 Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In Tom Conte and Yuanyuan Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, pages 577–589. ACM, 2016.

14 Alexei Colin and Brandon Lucia. Chain: tasks and channels for reliable intermittent programs. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 514–530. ACM, 2016.

15 Riccardo Dall'Ora, Usman Raza, Davide Brunelli, and Gian Pietro Picco. SensEH: From simulation to deployment of energy harvesting wireless sensor networks. In *IEEE 39th Conference on Local Computer Networks, Edmonton, AB, Canada, 8-11 September, 2014 - Workshop Proceedings*, pages 566–573. IEEE Computer Society, 2014.

16 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

17 Joakim Eriksson, Fredrik Österlind, Thiemo Voigt, Niclas Finne, Shahid Raza, Nicolas Tsiftes, and Adam Dunkels. Accurate power profiling of sensornets with the COOJA/MSPSim

simulator. In *IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, pages 1060–1061, 2009.

18 Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

19 Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213. ACM, 2019.

20 Graham Gobieski, Amolak Nagi, Nathan Serafin, Mehmet Meric Isgenc, Nathan Beckmann, and Brandon Lucia. MANIC: A vector-dataflow architecture for ultra-low-power embedded systems. In *IEEE/ACM International Symposium on Microarchitecture*, pages 670–684, 2019.

21 Josiah D. Hester, Timothy Scott, and Jacob Sorber. Ekho: realistic and repeatable experimentation for tiny energy-harvesting sensors. In Ákos Lédeczi, Prabal Dutta, and Chenyang Lu, editors, *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems, SenSys '14, Memphis, Tennessee, USA, November 3-6, 2014*, pages 1–15. ACM, 2014.

22 Matthew Hicks. Clank: Architectural support for intermittent computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 228–240. ACM, 2017.

23 Susan Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In Tony Montgomery, Lori A. Clarke, and Carlo Ghezzi, editors, *International Conference on Software Engineering, Melbourne, Australia, May 11-15, 1992*, pages 392–411, 1992.

24 Texas Instrument. MSP430FR599x Technical Documentation. https://www.ti.com/product/MSP430FR5994.

25 Hrishikesh Jayakumar, Arnab Raha, Jacob R. Stevens, and Vijay Raghunathan. Energy-aware memory mapping for hybrid FRAM-SRAM mcus in intermittently-powered iot devices. *ACM Trans. Embed. Comput. Syst.*, 16(3):65:1–65:23, 2017.

26 Mustafa Emre Karagozler, Ivan Poupyrev, Gary K Fedder, and Yuri Suzuki. Paper generators: harvesting energy from touching, rubbing and sliding. In *ACM symposium on User interface software and technology*, pages 23–30, 2013.

27 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization*, page 75, 2004.

28 Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices*, 50(6):575–585, 2015.

29 Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. Architecture exploration for ambient energy harvesting nonvolatile processors. In *IEEE International Symposium on High Performance Computer Architecture*, pages 526–537, 2015.

30 Kiwan Maeng, Alexei Colin, and Brandon Lucia. Alpaca: intermittent execution without checkpoints. *Proc. ACM Program. Lang.*, 1(OOPSLA):96:1–96:30, 2017.

31 Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 129–144, 2018.

32 Kiwan Maeng and Brandon Lucia. Supporting peripherals in intermittent systems with just-in-time checkpoints. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1101–1116, 2019.

33 Azalia Mirhoseini, Ebrahim M Songhori, and Farinaz Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *IEEE International Conference on Pervasive Computing and Communications*, pages 216–224, 2013.

34 The MSP430 emulator. https://github.com/contiki-ng/mspsim.

35 Benjamin Ransford, Jacob Sorber, and Kevin Fu. Mementos: System support for long-running computation on RFID-scale devices. In *ACM SIGARCH Computer Architecture News*, pages 159–170, 2011.

36 Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995.

37 Emily Ruppel and Brandon Lucia. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1085–1100, 2019.

38 Alanson P Sample, Daniel J Yeager, Pauline S Powledge, Alexander V Mamishev, and Joshua R Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement*, 57(11):2608–2615, 2008.

39 Claude E. Shannon. Communication theory of secrecy systems. *Bell Syst. Tech. J.*, 28(4):656–715, 1949.

40 Milijana Surbatovich, Limin Jia, and Brandon Lucia. I/O dependent idempotence bugs in intermittent systems. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):183, 2019.

41 Charles Suslowicz, Archanaa S Krishnan, and Patrick Schaumont. Optimizing cryptography in energy harvesting applications. In *Proceedings of the Workshop on Attacks and Solutions in Hardware Security*, pages 17–26. ACM, 2017.

42 Joel Van Der Woude and Matthew Hicks. Intermittent computation without hardware support or programmer intervention. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 17–32, 2016.

43 Jiawei Yuan and Shucheng Yu. Privacy preserving back-propagation neural network learning made practical with cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 25(1):212–221, 2013.