Towards A Power Advisor in a Devkit for Internet-of-Things Microcontrollers

Vincent Morice, Florence Maraninchi, Jérôme Cornet

Towards A Power Advisor in a Devkit for Internet-of-Things Microcontrollers

Vincent Morice*
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France
vincent.morice@univ-grenoble-alpes.fr
vincent.morice@st.com

Florence Maraninchi
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France
florence.maraninchi@univ-grenoble-alpes.fr

Jérôme Cornet
STMicroelectronics, 38019, Grenoble, France
jerome.cornet@st.com

ABSTRACT

Microcontrollers (MCUs) for the Internet-of-Things (IoT) are powerful and versatile computing platforms, which may be hard to program correctly and efficiently; power performance is particularly important. We inestigate automatic methods to detect software performance anti-patterns for this class of systems, so as to help the software developer with power-related aspects. We use a virtual prototype, i.e., we execute the real object code on a simulated model of the hardware platform, given as a transaction-level model (TLM) augmented with dedicated monitors. We study two cases taken from an industrial example, and show that our method can help detect patterns that would be difficult to detect statically, even when the source code is available, because they involve the state of the hardware and the timing of operations.

KEYWORDS

Transaction-Level Modeling, IoT, Power Consumption, DevKit

1 INTRODUCTION

1.1 IoT Chips

The STM32WB (STMicroelectronics), the CC2540 (Texas Instrument), the BM70 (Microchip), or the QN9080 (NXP), are powerful and versatile microcontrollers units (MCUs), containing one or more CPUs (like ARM-Cortex series) and integrated with various sensors, actuators, and connectors, designed with power performance in mind. They may be powered by a battery and offer sophisticated clock and power modes. The difficulty to write good software for those platforms is a known problem in the industry: software should exploit the capabilities of the hardware correctly (respecting its specification), efficiently (consuming as little power as possible), and securely (resisting attacks). In this paper we focus on power efficiency. Dealing with numerous sensors makes this type of HW/SW platform a cyber-physical system (CPS): some knowledge about the physical environment can help build efficient solutions, for instance by selecting appropriate refreshment rates for sensors. A thorough understanding of the computing platform is needed in order to guarantee power performance, because a very small variation in the software can have tremendous effects on the consumption, and therefore the lifetime of the IoT object. In one of the examples below (see 2.1.2), the lifetime increases from approximately 3 days to approximately 8 months with a 700 mAh battery.

*Also with STMicroelectronics, 38019, Grenoble, France.

1.2 Methods and Tools for the Software Developer

Difficulties to write efficient and correct software can be tackled in various ways. Let us consider the pros and cons of each solution.

(1) Careful reading of the source code by experts: requires high expertise, not necessarily available at the customer site, and access to the source code, not always available for libraries.

(2) Using some Hardware-Abstraction-Layer (HAL) provided by the vendor: helps avoiding bugs and writing portable code; but it is sometimes counter-productive as far as power efficiency is concerned.

(3) Using interactive debugging tools like GDB for the software running on the real chip: requires heavy human interaction and may fail to correct power consumption or timing problems because they are due to the invisible hardware states.

(4) Executing the software on some simulated (thus fully observable) model of the hardware: gives non-intrusive access to many details of the hardware platform that may be hard to observe on the real chip, while having a clear impact on power consumption, like the traffic on the bus or the power modes of the CPU; some vendors provide simulators for a partial view of their platform, like the configuration of the clocks.

(5) Applying data mining techniques to the automatic detection of patterns in execution traces [10]: relies on the availability of a large number of traces, and algorithms to search those large data sets; it is not meant for a monitoring (incremental) implementation.

(6) Automatically detecting software (anti-)patterns [14]: can identify critical points quickly in a large number of code lines, but these points are not necessarily bugs or actual problems in real executions; it was first proposed for object-oriented development; it is more difficult on less-structured low-level code.

1.3 Exploiting Virtual Prototypes

We focus on power performance: the MCU or the sensors provide several modes that influence power, which makes it very hard to talk about power at a syntactical level; we need to look at the dynamic behavior.

Our goal is to explore a solution of type (4) above. It relies on a virtual prototype of the platform, given as a Transaction-Level Model (TLM). Virtual prototypes in the form of TL models started almost 20 years ago as tools to help hardware and software developers communicate, and also to allow for developing software before the hardware is indeed available [8]; they are on the path to be a

key aspect of the IoT ecosystem, providing developers with tools that help accelerate software development [11].

Our ultimate objective is to provide a development kit based on the TL model of a platform, extended with performance diagnosis capabilities. This kit will automatically warn the user about inefficient uses of the hardware, at execution time. Our aim is not only to diagnose inefficient uses, but also to provide the user with some advice for better code, hence smoothing the learning curve.

1.4 Contributions and structure of the paper

In Sections 2 and 3 we study a weather station based on a STM32F411 chip [5]. The code is a refactoring of a representative real industrial example from STMicroelectronics and partially uses the hardware abstraction layer provided by the company. The chip is mounted on a Nucleo-64 prototyping board [3] extended with a sensor expansion board including various sensors [1]. We identify two classes of very common problems to be detected by monitors: polling loops, and inefficient use of sensors. we show why dynamic information is key to the detection of *real* problems, and we show how to choose specific events and patterns to observe in TL traces in order to find occurrences of those two problems. In Section 4 we give a quick view of the monitor implementation, and evaluate our solution. Sections 5 and 6 give related work and conclude.

2 CASE STUDY: POLLING LOOPS

The first example is about detecting polling loops because it is generally more costly than waiting for an appropriate interrupt if available. When using methods based on the structure of the code, it means detecting the pattern `while(condition){};`, either syntactically, or on the binary code. As far as power consumption is concerned, the cost comes from the CPU being active (while it could be sleeping), but also from the bus activity induced by the evaluation of the condition. We aim at detecting polling loops even if the corresponding code is spread across several layers of low-level code, part of it possibly available in binary form only. Moreover, we would like to warn only when they have a significant effect, which may depend on the effective number of passes in the loop: looping twice is probably harmless, but 50 times may be significant.

We give two examples: the first one has to be replaced by something more efficient; the second one can be kept as it is.

2.1 Example: a Delay implemented by a Polling Loop

2.1.1 Existing Code. In the first example, polling is used to wait for a certain delay, as shown on Figure 1. `HAL_Delay()` takes as a parameter the time to wait (`Delay`), saves the current time and starts the polling loop. `uwTick` is a global variable incremented on every `SysTick` timer interrupt. The timer is configured to fire its overflow interrupt every millisecond. This example is used in a weather station application whose `main()` function is shown on Figure 2: it reads the sensors, then prints the value using the UART and then waits for 800 ms using the `HAL_Delay()` function.

2.1.2 Proposed Modification. Implementing the wait functionality efficiently on our platform is easy: put the MCU in a deep sleep mode and use the Real Time Clock (RTC) to wake it up. The RTC behaves as a timer, sending an interrupt after counting up to a certain value. The delay function now simply launches the count and puts the chip in STOP mode. The RTC also has to be initialized and its interrupt handler has to acknowledge the interrupt.

```
1 void HAL_Delay(__IO uint32_t Delay) {
2   uint32_t tickstart = 0;
3   tickstart = HAL_GetTick();
4   while((HAL_GetTick() - tickstart) < Delay) {}
5 }
6 uint32_t HAL_GetTick(void) { return uwTick; }
7 void HAL_IncTick(void) { uwTick++; }
8 void SysTick_Handler(void) { HAL_IncTick(); }
```

Figure 1: A delay implemented by a polling loop

```
1 int main(void) {
2   init();
3   while(1) {
4     int16_t temp = EnvSensor_GetTemperature();
5     int32_t pressure = EnvSensor_GetPressure();
6     uint16_t humidity = EnvSensor_GetHumidity();
7     printf("T, P, H = %d\n",temp,pressure,humidity);
8     HAL_Delay(800);
9 }}
```

Figure 2: `main()` function of the weather station application

```
1 /* asking access to registers */
2 CLEAR_BIT(RTC->CR, RTC_CR_WUTE);
3 /* Wait till access to wakeup timer is allowed */
4 while(READ_BIT(RTC->ISR,RTC_ISR_WUTWF) != 1);
5 config_wakeup_timer();
```

Figure 3: Polling loop on registers of the wakeup timer

The STOP mode disables a lot of peripherals, including the MCU, the SysTick timer and the bus, in order to save power. According to the ST tool "STM32 CubeMX" [4], which gives approximate power consumption related to those MCU states, implementing the wait process using the RTC and the STOP mode instead of the polling loop decreases the consumption from 8.5 mA to 0.12mA. When powered by a 700 mAh battery, the code modification can extend the lifetime of the system approximately from 3 days to 8 months.

2.2 Example: Unlocking Registers

2.2.1 Existing Code. The second example (Figure 3) is a piece of code used to *unlock* access to the configuration registers of the wake-up timer. The manual [5] explicitly requires the software to implement this polling loop. The loop waits for the Wakeup Timer Writing Flag (`WUTWF`) to be set, indicating that other configuration registers can now be written. The Wakeup timer is a feature of the RTC component, so the left part of the condition checks for the `WUTWF` bit of the RTC Initialization and Status Register (`ISR`) using the macro `READ_BIT(RTC->ISR,RTC_ISR_WUTWF)`.

In the real platform the wakeup timer has its own clock, different from that of the CPU, and the unlocking operation takes a few cycles of the wakeup timer's clock. The effective number of iterations depends on the relative speeds of the CPU's clock and the wakeup timer's clock. The faster the CPU's clock, the more loop iterations can be executed during the time it takes to perform the unlocking operation.

2.2.2 Keeping the Polling Loop. This is a typical case of a harmless polling loop because the way the hardware is designed ensures a very small number of iterations (in our example, we observed 20 in the worst case). We use transaction-level models, which can be approximately timed with realistic assumptions on the duration of transactions. Hence simulations with such models also show a small number of iterations. This type of polling case can be ignored easily, by using a threshold on the number of iterations.

2.3 Characterizing and Detecting Polling Loops on TL Models

The first step is to characterize problematic polling loops by observing the dynamic behavior of the code. The idea is to focus on the *actual effect* of a polling mechanism, even if it is not written as a typical polling loop; or it is, but with harmless effects.

The effect of a polling mechanism is characterized by some repetitive pattern being observed, and we can decide to issue a warning after a given number of repetitions. We propose to observe the sequence S of READ transactions issued on the bus. Section 4 explains what is indeed observable (not all variable accesses do generate bus transactions), and how to filter out other READ transactions (e.g., from interrupt handlers).

In the first example above, there are 3 variables in the condition of the loop: `Delay`, `uwTick` and `tickstart`. The latter is stored in a CPU register, hence does not generate bus transactions. There are only two addresses that appear in READ transactions; (`HAL_GetTick()` function calls also generate READ transactions, but this is not an issue as explained in section 4.) `uwTick++` is filtered since it is executed in an interrupt handler.

In the second example there is only one access to a register of the RTC component that generates a READ transaction on the bus. Section 4 explains the instrumentation of the TL model, and shows how to detect the first case, while ignoring the second one.

The last important point is to characterize the repetitive patterns of READ transactions that are indeed issued on the bus for typical polling loops. For instance, if the code indeed contains a loop `while(condition){};`, each evaluation of the condition generates successive READ transactions, depending on the logical structure of the condition (in C the evaluation of `cond1 && cond2` does not evaluate `cond2` — hence does not access its variables — if `cond1` is false). There is no cache between the CPU and the bus, so each access to a variable generates a transaction. Observing repeated accesses to the variables of `cond1` is a hint that some polling situation might be involved, but it depends on which other accesses are observed between the accesses to the variables of `cond1`. Extracting also the variables of `cond2` helps confirm that there is a polling case.

2.4 Formalizing Patterns

As an example, let us consider the dynamic behavior of a program that checks repeatedly a condition `cond`:
`((a < 12)&&(a >= 0))||(b == 0)||(c == 0)`,
either written exactly like that, or obtained with macros and calls to other software layers. Notice the same variable may appear several times in the condition.

The iterative evaluation of `cond` generates sequences of accesses of the form `(a|aa|aab|aabc)`* if all variables are observable (i.e., generate transactions on the bus) or simply `(a|aa|aac)`* if, e.g., `b` does not.

Since we do not know in advance which addresses `a, b, c` to look for when searching for polling loops, our problem is a *parametric* version of the above example: we search for `(x|xy|xyz|xyzt|...)`*, where `x, y, z, t, ...` can be instantiated with any address. The number of these "parameters" depends on the condition. It is not too restrictive to consider that it is *bounded* by a relatively small number P. In the sequel, we take $P = 3$ as an example (3 observable accesses). The problem becomes to check whether an execution trace is of the form: `(x|xy|xyz)`* for some `x, y, z`. For a given vocabulary V and $m \in V^*$, we can define $m \in$`(x|xy|xyz)`* as: $\exists (a,b,c) \in V^3 . m \in$`(a|ab|abc)`*. If the vocabulary V where `a, b, c` belong is finite, this can be written as a simple regular expression, because $\exists$ can be expanded into ordinary alternatives: $m \in$`(x|xy|xyz)`$^* \iff m \in \bigvee_{(a,b,c)\in V^3}$`(a|ab|abc)`*.

Finally, we replace the * by $^{[n,+\infty]}$, to start warning about the presence of a polling loop only if it exceeds a number n of effective iterations. In order to validate quickly the idea of using monitoring techniques to detect polling loops, we implemented a simpler (yet very frequent) case without Boolean operations. Instead of searching instances of `(x|xy|xyz)`$^{[n,+\infty]}$, we search for `x`$^{[n,+\infty]}$ `|` `(xy)`$^{[n,+\infty]}$ `|` `(xyz)`$^{[n,+\infty]}$. See details in Section 4.

This example does not fully characterize all polling loops. For instance, instructions in the condition or the loop body (especially branching) can partially hide the repetition or produce a large number of repetitive addresses observed on the bus. Moreover, if we choose a very big P, we might end up detecting the infinite loop of the main program. However we think this is a promising approach, since it already detects a lot of typical polling loops in typical code samples. Further work will include more cases.

3 CASE STUDY: TEMPERATURE SENSOR

The second case-study concerns the use of the LPS22HB temperature sensor [2], which has several operating modes: (1) In **one-shot** mode the software has to set the one-shot bit to ask for a new value to be measured and prepared; the bit will be cleared by hardware when the new value is available; (2) In **auto-refresh** mode the sensor can produce a new value with a given period chosen in $\{13, 20, 40, 100, 1000\}$ milliseconds. If the application needs less than one value per second, or more than one every 13 ms, it should use the one-shot mode. In both modes the last measure is always available so that the software can read the value at any time.

The idea of this example is to detect the uses of the one-shot mode that could be improved by using the auto-refresh mode instead. The one-shot mode can have a significant impact on the consumption, because it involves more transactions on the MCU bus: (1) the software sends the address of the control register of the sensor and the data ("1" on the one-shot control bit), (2) it sends the address of the temperature data register in order to read the value. The auto-refresh mode is far more efficient, since only step (2) is needed. Moreover a READ needs two distinct accesses to two 8-bit registers containing the MSBs and the LSBs of the temperature value.

3.1 Example

3.1.1 Existing Code. Our application uses the one-shot mode (see 4). The bit is set (`Set_One_Shot`), requiring a new measure; next time the function will be called, it will access the value (`Get_Temp`) and get this new measure. In our example, the function is called approximately each 800ms (see main code on Figure 2).

```
int16_t EnvSensor_GetTemperature(){
    int16_t SensorValue;
    if(TEMP_SENS_IsInitialized()){
      /*Read the previous value of the sensor
        and restart the One Shot for the
        next measurement*/
      if(TEMP_SENS_Get_Temp(&SensorValue)){
        if(TEMP_SENS_Set_One_Shot()){
           return SensorValue;}}}
    return -1;} // Error
```

Figure 4: Temperature measurement code in one-shot mode (simplified).

```
int16_t EnvSensor_GetTemperature(){
    int16_t SensorValue;
    if(TEMP_SENS_IsInitialized()){
      if(TEMP_SENS_Get_Temp(&SensorValue)){
         return SensorValue;}}}
    return -1;} // Error
```

Figure 5: Temperature measurement code in auto-refresh mode (simplified).

3.1.2 Proposed Modification. The alternative implementation is to use the auto-refresh mode, as shown on Figure 5. The auto-refresh period is set in the `init()` function called at startup in the `main()` function. The idea is to choose a period close to the delay D implemented by the application ($800ms$ in our case). If we do not have the full source code, or because it is hard to analyze statically anyway, D can be estimated dynamically by looking at the TL traces. On this example we choose $100ms$.

The cost of one bus transaction alone is very hard to estimate, so the power consumption of the `main()` function (Fig. 2) has been measured on the board with the initial and modified versions of the `EnvSensor_GetTemperature()` function and our proposed modification for the `HAL_Delay()` function (see section 2.1.2). Average consumption is 1.57 mA (± 0.01 mA) for auto-refresh mode and 2.47 mA (± 0.01 mA) for one-shot mode. It would correspond to an increase in the lifetime from approximately 12 days to more than 18 days with a 700 mAh battery.

3.2 Detecting the One-Shot Mode on TL Traces

The type of situation we want to detect involves software intended to read a fresh value of the temperature at a regular rate. We assume the one-shot mode is always less efficient than the auto-refresh mode when used with a period $P_r < 1$s. P_{max} and P_{min} are the respective maximum and minimum possible auto-refresh periods of the sensor (here $P_{max} = 1$s and $P_{min} = 13$ms). The decision is given by Table 1. The last line shows a hypothetical case where the software needs a new value more than once every 13 ms. It might be useful for some critical applications where the temperature is rising up extremely fast, but it cannot be accomplished in auto-refresh mode, the one-shot mode should be used. In this case this is not a matter of power efficiency so this is out of our scope.

We can observe the accesses to the sensor registers: reading the value, and requesting the one-shot mode. The tricky part is to measure time between requests to estimate the period, because we work with a simulated model of the hardware platform for which timing is always an approximation; moreover, even with a simple loop code, the software on the real platform does not read the value on a strictly periodic way.

We measure the period several times and compute the average.

Table 1: Detecting Inefficient Uses of the Sensor

	One-shot	Auto-Refresh
$P_{max} > P_r > P_{min}$	inefficient (too many transactions)	efficient
$P_r > P_{max}$	efficient	inefficient (some values will be overwritten before they are read)
$P_r < P_{min}$	correct	wrong (some values will be read twice)

4 IMPLEMENTATION AND EVALUATION

4.1 The TL Platform

The virtual prototype is implemented as a SystemC TL model of the STM32 MCU, including an instruction-set-simulator provided by ARM, and a TL model of the LPS22HB temperature sensor. For the first example (polling loops) we observe bus transactions. Direct Memory Interface (DMI) has to be disabled to ensure transactions visibility. For the second example, we observe the transactions on some registers of the sensor: `CTRL_REG2` is written when a value is asked in one-shot mode, `TEMP_OUT_H` and `TEMP_OUT_L` contain the temperature value, `CTRL_REG1` controls the auto-refresh rate.

4.2 How to Add Monitors

In our proof-of-concept implementation, the monitors are first implemented as classes in Python scripts. The scripts are called at the beginning of the SystemC simulation, and instantiate Python objects representing the monitors. Using Python allows high flexibility and coding simplicity: adding a new monitor can be done without heavy C++ re-compiling of the platform.

In the TL model, the bus and the sensor registers are SystemC modules that receive transactions. The object constructor of the monitors sets a watchpoint on a SystemC module and is programmed to trigger when the module receives a transaction with certain conditions, which may involve the metadata of the transaction. When it happens, the SystemC simulation is paused, the state of the hardware model can be inspected (for instance bits in control registers), the simulation date and all the metadata of the transaction are given to a Python method of the monitor. When the method returns it gives control back to the SystemC simulation.

The monitors can wait for a user action before giving control back to the SystemC simulator. This can be useful for the user to see the current instruction of the embedded software, via a connected debugger, or to inspect the internal state of the monitors.

4.3 Monitor for the Detection of Polling Loops

Observing the transactions on the bus is always available in TL models, but not necessarily the values of the registers internal to the CPU, like the program counter, if the CPU model is vendor-specific. So our polling loop detection works only with the memory accesses and the accesses to registers in components distant from the CPU. The monitor sets a watchpoint on the bus that triggers on the condition READ. When SystemC gives control to the Python script, it also transmits the target module name and the address.

4.3.1 Filtering Bus Transactions. We need to observe the sequence S of read addresses issued on the bus by running the polling loop alone. But other accesses are due to the execution of the interrupt handlers on the same CPU (or even other threads if some scheduler was used, but we consider bare-metal implementations). It is possible to ignore any transaction issued while an interrupt is in *active* state. The virtual prototype includes a model of the nested vector interrupt controller (NVIC) that provides a register to indicate whether any interrupt is active or not. An interrupt is active when the CPU is running the interrupt handler (see STM32 reference manual [5]). As the first monitor is disabled when an interrupt is active, we instantiate another monitor — destroyed at the end of the interrupt handler — in order to detect polling loops inside it. So the maximum number of monitor instances is the number of possible interrupts: 255. Transactions due to fetching instructions from memory can also be filtered out: the memory location of the code is known so the corresponding addresses can be ignored.

4.3.2 Recognizing a Polling Loop Sequence. We now work only with READ transactions observed on the bus, targeting the data memory section and the peripherals. As mentioned in section 2.4, the monitor looks for the regular expression $\mathtt{x}^{[n,+\infty]}$ | $(\mathtt{xy})^{[n,+\infty]}$ | $(\mathtt{xyz})^{[n,+\infty]}$ in the sequence of transactions. The recognizer is fully implemented in Python in the monitor class, initialized with a given maximal size of the repetitive patterns to be recognized P, and a minimum number of iterations n. We use $P = 3$ and $n = 21$ in the example.

4.4 Monitor for Sensor Mode Advice

We consider that the user provides our tool with the reference of the sensor used (here LPS22HB); the tool can instantiate a corresponding TL model enriched with the detection of the available refresh periods.

The monitor sets three watchpoints: (1) On the one-shot bit of the register `CTRL_REG2` triggering on the condition WRITE, (2) on the register `TEMP_OUT_L` triggering on the condition READ and (3) on the register `TEMP_OUT_H` with the same condition. When a watchpoint is triggered, the monitor updates the associated measured periods with the simulation date. It then checks for the inequalities of the table 1 and warns the user in the inefficient cases. The monitor also checks the field `ODR` of the register `CTRL_REG1` that indicates the sensor mode (`ODR` $\neq 0$ for auto-refresh, 0 for one-shot).

4.5 Example Results as Shown to the Developer

In our example, Fig. 6 is the message displayed when the monitor detects an inefficient use case of the sensor. If a debugger is connected then the simulation is stopped and it indicates that the software is currently running the `TEMP_SENS_Set_One_Shot()` function in the call stack. Fig. 7 is the message for a polling loop. The debugger indicates that the software is currently running `HAL_Delay()`.

```
Possible inefficient use of the sensor:
top.NODE_0.LPS22HB.registers.CTRL_REG2
ONE-SHOT asked with a period: 803,135,151 ns.
auto-refresh at 100 ms might save some transactions.
```

Figure 6: Detection of Inefficient Uses of a Sensor

```
Polling suspected at addresses 0x20000260, 0x20017FD0
target: top.NODE_0.NUCLEO.STM32.RAM
You might consider putting the CPU in sleep mode and
programming a wake-up interrupt!
```

Figure 7: Detection of a Polling Loop

4.6 Accuracy of the Detection Principle

The `Hal_Delay()` polling loop (Fig. 1) is detected correctly after 21 iterations and the message of Fig. 7 is displayed. Our proposed modification using the RTC wakeup timer, running the harmless loop (Fig 3) is filtered out correctly with the same monitor. Other polling loops are also successfully detected like the example of Fig. 8, although they include some code inside the loop which generates other transactions on the bus.

```
1  /* Wait until ADDR or AF flag are set */
2  tmp1 = __HAL_I2C_GET_FLAG(hi2c, I2C_FLAG_ADDR);
3  tmp2 = __HAL_I2C_GET_FLAG(hi2c, I2C_FLAG_AF);
4  tmp3 = hi2c->State;
5  while((tmp1 == RESET) &&
6     (tmp2 == RESET) &&
7     (tmp3 != HAL_I2C_STATE_TIMEOUT)){
8       if((Timeout == 0) ||
9         ((HAL_GetTick() - tickstart ) > Timeout))
10          hi2c->State = HAL_I2C_STATE_TIMEOUT;
11      tmp1 = __HAL_I2C_GET_FLAG(hi2c, I2C_FLAG_ADDR);
12      tmp2 = __HAL_I2C_GET_FLAG(hi2c, I2C_FLAG_AF);
13      tmp3 = hi2c->State;}
```

Figure 8: A Complex Polling-Loop Example.

4.7 Impact of the Instrumentation on Simulation Time

In all cases above, the embedded software has been compiled with the `-O0` option using embedded GCC. The additional cost of the instrumentation is due to: (1) the context switches between SystemC and Python; (2) the fact that we have to disable the Direct-Memory-Interface (DMI) optimization to ensure visibility of transactions; (3) the cost of the pattern detection algorithms themselves. The cost of detecting sensor uses is negligible, because the registers are accessed very seldom in the main loop, and checking the sensor mode does not require any costly operation.

We therefore focus on the polling-loop monitor. We measured the simulation duration from the start until the first call to `HAL_Delay(800)` in the main function (Fig. 2) with various combinations of DMI and monitor enabling and disabling. In table 2, *None* indicates that the monitor is disabled. *C++* indicates that the

Monitor		*None*	*C++*	*Empty*	*Present*
DMI	*On*	0.7 sec	0.73 sec	2.82 sec	3.06 sec
	Off	1.19 sec	1.24 sec	24.05 sec	28.41 sec

Table 2: Impact of enabling/disabling DMI and monitor during simulation, measured until first call to `HAL_Delay()`.

monitor is implemented directly in the model in C++, *Empty* indicates a monitor that triggers a SystemC to Python context switch at each READ bus transaction, but does nothing. This is useful to evaluate separately the cost of the pattern detection algorithm. The DMI is disabled (*Off*) only on the data section memory zone, hence code fetching remains invisible.

The duration have been measured using the "`time`" linux command, adding the user and the system times. Results show a serious performance breakdown due to Python/SystemC context switches, but this can be avoided writing the monitor in C++. The pattern recognition algorithm itself and disabling the DMI also slow down the simulation. In addition leaving DMI enabled, and instantiating monitors, also have a small impact while it should not (DMI enabling hides all memory bus transactions, so the watchpoint never triggers). This is due to the loading of the python monitor when simulation starts.

These measures show that we can focus on the detection algorithms alone, the instrumentation mechanism being sufficiently efficient.

5 RELATED WORK

[12] proposes to use dynamic binary instrumentation (DBI) to detect *excessive dynamic memory allocations*, and argues that it is a software performance anti-pattern very difficult to detect statically, because it relies on timing (it detects short-lived, high-frequency dynamic memory allocations). The general framework is very similar to ours. We use a simulation of the hardware because power consumption depends on the *state* of the hardware platform, and sometimes on timing. We may use instrumentations of an instruction-set simulator, which has the same potential as DBI. [12] also provides an interesting review of other dynamic approaches for other *software performance anti-patterns*.

[9] is able to detect certain software performance anti-patterns (resource leaks such as CPU, memory, battery) in android applications, which are sometimes imposed by the underlying frameworks. In our case, this would be due to the HAL. The approach is based on analysing the code of the application, thus being static but without the need for the source code. The 8 patterns detected are all related to the static structure of the code.

[10] applies data mining techniques to the analysis of real-time streams in multimedia applications. It helps understand the violations of QoS properties, due to tasks missing their deadline. The implementation is not meant for monitoring contexts, and may need the full trace.

[7] presents efficient algorithms to find *frequent* sequences in databases of ordered transactions. The type of patterns that may be searched resembles what we need for the polling case, but the search criteria is a quantitative measure of the frequency in the whole database. On the contrary, our definition of a polling is local, it does not relate to the frequency, in the whole behavior, of the memory transactions generated by the evaluation of the loop condition. Moreover, the family of algorithms developed for pattern mining in databases does not necessarily work in a *incremental* way, which is necessary for our *monitoring* purpose.

6 CONCLUSION AND FURTHER WORK

We showed on two frequent examples how detecting problems dynamically allows to focus on the real impact of bad software. A piece of code that looks like a polling loop is not always one, and even if it is, it is not always bad for energy consumption. The replacement of the polling loop involves the use of a CPU sleep mode, whose effect cannot be captured at source level. For the sensor example, the dynamic detection on a timed simulation model allows to reason about periods, which, again, would be difficult statically. Further work will be devoted to other classes of power-related problems. Ongoing work is devoted to the design of a dedicated algorithm for the detection of polling loops, exploiting the fact that the pattern `(x|xy|xyz)` has a very particular shape. We will also investigate whether our patterns can be formalized as properties of the traces written in languages like PSL [6], so that they can then be *compiled* into monitors, using techniques similar to those of [13], for instance.

REFERENCES

[1] [n.d.]. IKS01A2 Extension Board. www.st.com/en/ecosystems/x-nucleo-iks01a2.html.

[2] [n.d.]. LPS22HB Temperature Sensor. www.st.com/en/mems-and-sensors/lps22hb.html.

[3] [n.d.]. Nucleo F411RE Board. www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-nucleo-boards/nucleo-f411re.html.

[4] [n.d.]. STM32CubeMX. www.st.com/en/development-tools/stm32cubemx.html.

[5] [n.d.]. STM32F411RE. www.st.com/content/st_com/en/products/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus/stm32-high-performance-mcus/stm32f4-series/stm32f411/stm32f411re.html.

[6] Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari. 2004. *Using PSL/Sugar for formal and dynamic verification: Guide to Property Specification Language for Assertion-based Verification.* VhdlCohen Publishing.

[7] Minos N Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 1999. SPIRIT: Sequential pattern mining with regular expression constraints. In *VLDB*, Vol. 99. 7–10.

[8] Franck Ghenassia. 2005. *Transaction Level Modeling With SystemC: TLM Concepts And Applications for Embedded Systems.* Springer-Verlag.

[9] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Detecting antipatterns in android apps. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. IEEE Press, 148–149.

[10] Oleg Iegorov, Vincent Leroy, Alexandre Termier, Jean-François Méhaut, and Miguel Santana. 2015. Data Mining Approach to Temporal Debugging of Embedded Streaming Applications. In *Proceedings of the 12th International Conference on Embedded Software (EMSOFT '15)*. IEEE Press, Piscataway, NJ, USA, 167–176. http://dl.acm.org/citation.cfm?id=2830865.2830884

[11] Philippe Magarshack. 2018. Accelerating IoT Device Development — from Silicon to Developer Tools, Keynote Speech. In *DVCon Europe*.

[12] Manjula Peiris and James H Hill. 2016. Automatically Detecting Excessive Dynamic Memory Allocations Software Performance Anti-Pattern. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ACM, 237–248.

[13] Laurence Pierre and Luca Ferro. 2008. A tractable and fast method for monitoring SystemC TLM specifications. *IEEE Trans. Comput.* 57, 10 (2008), 1346–1356.

[14] Connie U Smith and Lloyd G Williams. 2000. Software performance antipatterns.. In *Workshop on Software and Performance*, Vol. 17. Ottawa, Canada, 127–136.