Adaptation of an embedded architecture to run Hyperledger Sawtooth Application

Roland Kromes
Université Côte d'Azur
LEAT / CNRS UMR 7248
Sophia Antipolis, France
roland.kromes@univ-cotedazur.fr

Luc Gerrits
Université Côte d'Azur
LEAT / CNRS UMR 7248
Sophia Antipolis, France
luc.gerrits@etu.univ-cotedazur.fr

François Verdier
Université Côte d'Azur
LEAT / CNRS UMR 7248
Sophia Antipolis, France
francois.verdier@univ-cotedazur.fr

***Abstract*—We want to study the adaptability of blockchain technology on IoT devices. In this paper we studied Hyperledger Sawtooth adaptability on a Raspberry Pi 3 B+ depending on the requirements of a special use case described by our industrial partner Renault. In addition to the study of Hyperledger Sawtooth adaptability, we modified the model of the BCM2837 architecture on SystemC-TLM. This model is known as the "heart" of the Raspberry Pi 3 B+. By our knowledge, this work conducts to the first IoT architecture able to interact with others by using a Hyperledger Sawtooth blockchain. By using this modification we could also obtain a significant gain, an acceleration of 112 for SHA-256 and 293 for SHA-512 cryptographic algorithms.**

***Index Terms*—Blockchain implementation in IoT embedded systems, Blockchain in low power IoT communication networks, Smart Contracts, Low Power Consumption Hardware Modeling**

I. Introduction

The history of Internet of Things (IoT) started at the early 00's thanks to the new type of applications using objects that were connected to the Internet [1]. The usage of IoT devices still increases, by statistics today the number of devices taking part of IoT takes almost 30 billion [2]. Most of these objects are constrained devices, that means that their hardware resources are limited.

The effective execution of new IoT applications depends on two basic exceptions: the computational power and the power consumption of the target IoT device. Our aim is to model an IoT device that is able to execute blockchain applications with a constraint of low power consumption and with a high performance. In our work we use an already existing hardware model BCM2837 (better known as the heart of the Raspberry Pi 3 B+) modeled on SystemC-TLM [3]. SystemC is a language used in the first steps of hardware design of such IoT devices. This programming language is popular in industry field when we design the really first solution of our architectures. The use of SystemC is relatively easy because this language is based on C++, thus is an object oriented language. In this paper we study the Hyperledger Sawtooth blockchain [4], next we modify the BCM2837 model to be able to execute Sawtooth applications with Smart Contracts and we increase its performance by adding special Intellectual Properties (IPs).

Since the last decade we see that the blockchain technology and especially the Smart Contracts are starting to be used in more and more use cases and in different domains. Today the use cases can be found not only in finance domain but even in healthcare, utilities, real estate domains and government sector [5]. In 2016 [6] there were 18 use cases and only 4 of them were created for IoT applications. Today, the IoT implementation of blockchains seems to be one of the first activity sectors in the blockchain research works.

We can notice that blockchain technology appears even in Smart City applications including the problematic of IoT device security [7]. It's clear that the main goal of the usage of blockchain with Smart Contracts is the high security and the tractability that this technology can provide. In our Smart IoT for Mobility project [8] we try to develop IoT devices that will be used in vehicles for connecting them on the blockchain.

A. Smart IoT for Mobility (SIM)

This multidisciplinary project is based on a new vehicle infrastructure that was imagined by our industrial partner Renault. In this new use case all of the car are connected to a blockchain. The cars are equipped with different types of sensors sending data about the state of the car, in different types of situation as an accident or car selling/buying. The sensors could be odometers (measure the mileage), radars (detecting the safe-distance), 360° cameras (sending photos about the environment of the car after an accident) and so on. Despite of the different use cases the main goal of Renault is to send the data after an accident. The data will be sent in a format of Smart Contracts on a blockchain not only to Renault but to the Police, to the insurance companies and eventually to a car repair shop. With the Smart Contracts the judgment procedure by the Police and insurance procedure could be faster, and all of the information about the accident would be stored for ever, because once the data is stored on a blockchain it become immutable. The accident use case of Renault is illustrated in the Fig. 1.

We have to notice that there are four academic research teams of different domains working on this project, domains are electronic engineering, computer science, economics and

978-1-7281-2530-5/19/$31.00 ©2019 IEEE

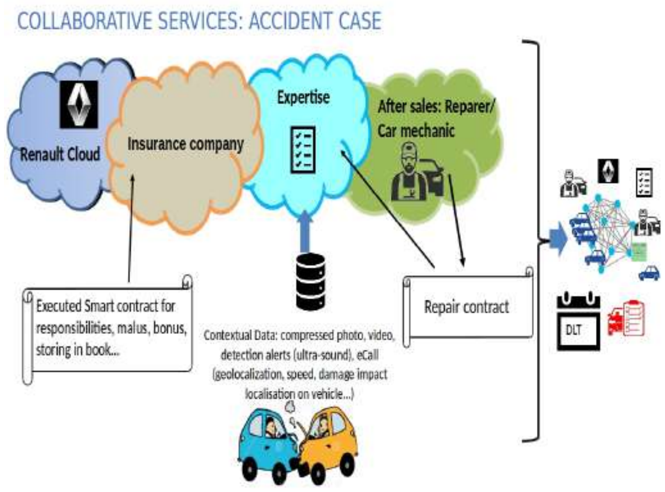
COLLABORATIVE SERVICES: ACCIDENT CASE
Renault Cloud
Insurance company
Expertise
After sales: Reparer/ Car mechanic
Executed Smart contract for responsibilities, malus, bonus, storing in book...
Contextual Data: compressed photo, video, detection alerts (ultra-sound), eCall (geolocalization, speed, damage impact localisation on vehicle...)
Repair contract
DLT

Fig. 1. Use case of Renault

low. The laboratories of these teams are members of the Université Côte d'Azur (UCA). Every team works on a different thread of the project. Our goal is to model a low power consumption IoT architecture for automobiles with the availability to send blockchain transactions as the application in the use case of Renault.

II. State of the Art

The blockchain technology starts to be more and more popular, we can notice the appearance of a new type of use cases and a need for merging this technology to IoT world. In [7] we can read about ideas how to deploy new smart cities with blockchain and IoT solutions.

A. Blockchain

In a blockchain the data is distributed between all of the members of the network. That means that every member contain exactly the same data. By this method we obtain a database more secured opposed to the server based solutions, because if a potential attack would make changes in a database it should be done in all of the member's database.

The blockchains are clustered in three big classes: public, consortium and private blockchain.

1) Public blockchain: is a decentralized network because every node can participate in the consensus process, and every node can read every type of data.

2) Consortium blockchain: is partially centralized. There is only a group of nodes that can take part of the consensus. And not every node has permission to read data.

3) Private blockchain: is a fully centralized network, the nodes participating in consensus and having permission to read are predefined by the organization that deployed this network [9].

In our project we are interested in private blockchains (in particular Hyperledger Sawtooth) because of our industrial partner Renault.

4) Blockchain's structure: basically a block of the blockchain contains the list of transactions and the hash of the previous block. The hash gives the reference of the previous block so in this way the blocks creates a chain and this structure provides a full tractability of transactions and Smart Contracts with a special time stamp [5] [9]. This structure is shown in Fig. 2

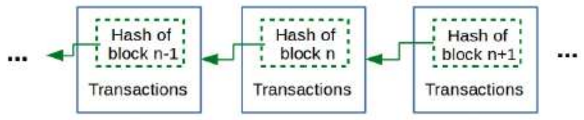

Fig. 2. Example of a blockchain

B. Smart Contracts

Smart Contracts are digital programs that could be written on different languages as Solidity, Java-Script and so on [10]. Smart Contracts were proposed by a computer scientist Nick Szabo in 1996. A traditional contract is deployed between two parties and it is verified with a trusted third party. Smart Contracts allows to avoid this trusted third party because one the contract is sent on a blockchain it becomes unchangeable. By using Smart Contracts signing procedures can be faster and more secured thanks to the secure characteristics of blockchain.

C. Example of Consensus

Consensus is a main part of the blockchain, because this algorithm provides the validity of a transaction. The consensus algorithm is solve by the minors of the network. The most popular consensus are described below:

1) Proof of Work (PoW): consensus is used in public blockchains. The idea is that the first miner node whom can solve a given cryptographic problem between the other miners will got the law to create a block (and it wins cryptocurrency). Ethereum blockchain use this type of consensus [11]. The adaptation of this type of consensus on an IoT is very difficult because of its resource needs [6].

2) Byzantine Fault Tolerance (BFT): is rather used in private blockchains than in public ones, because in this blockchains minors are not used for cryptocurrency gaining but for obtaining a secured network. In BFT if 3/2 of minors are agree with the validity of the transaction then it would be validated and it would be taken into a block [12]. This consensus is used for example in Hyperledger Fabric [13].

3) Power of Elapsed Time (PoET): is mainly used in Hyperledger Sawtooth. We explain this consensus below.

D. Hyperledger Sawtooth

Hyperledger Sawtooth is an enterprise blockchain platform, thus it is a permissioned (private) blockchain. Sawtooth was designed to achieve a secured, scalable and modular structure. Its structure is modular because it contains modular consensus

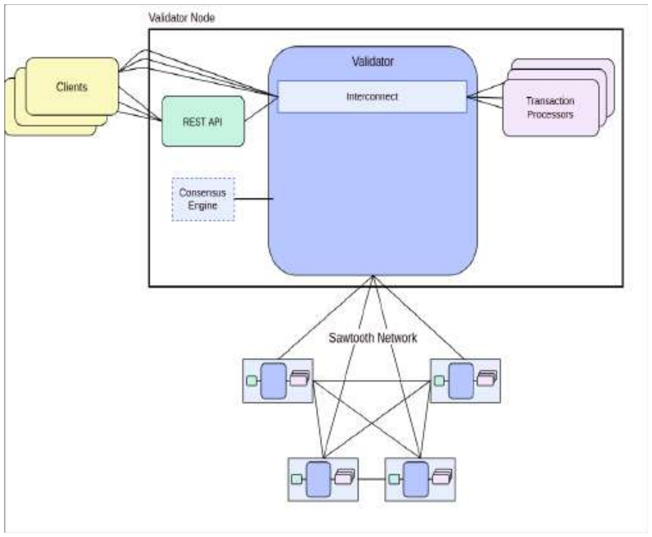

Fig. 3. Sawtooth's architecture [15]

that means that Sawtooth is able to use different type of consensus as PoW, PBFT, and PoET (Proof of Elapsed Time). In addition to the consensus, Sawtooth is modular because, it is possible to implement own modules in a node interacting with the Validator (that could be seen as the miner in Ethereum blockchain). A big advantage of Sawtooth is that the consensus can be changed on running time [4].

Basically Sawtooth uses the consensus of Proof of Elapsed Time that was invented by Intel for Software Guard Extensions (SGX) that offers hardware-based security encryptions [14]. We have to notice that this consensus can be used not only on SGX hardware specific architectures. In other type of architectures PoET will be simulated. In our experiments we have equally used this consensus.

1) PoET consensus: this consensus is an implementation of the PoW to be able to use in a private blockchain. Contrarily to PoW's goal where fastest node creates block after solving a hard cryptographic problem, here the node can create a block if it has the smallest time sample that was distributed. In this rule the node doesn't need to work all the time as in PoW. Since the node doesn't have to solve a hard problem as fast as it is possible, in this blockchain the hardware resource can be less developed than in public blockchains. Thus theoretically Sawtooth network consumes less energy and could be adapted easier on IoT devices [4] [15] [16].

2) Architecture of Sawtooth: as we can see on Fig. 3 a Sawtooth network is constructed by Validator nodes (we can consider them as miners in other type of blockchains). The interacting with Validator can be done with Client application via a REST API. It has to be noticed that Transaction Processors and the Consensus Engine are modules that are outside of the Validator, that is the reason why we are able to change these modules easily [15].

The deployment of Smart Contracts is possible on Sawtooth, the Smart Contracts are called Transaction Processors (TPs). These Transaction Processors can be written on any programming languages as Python, Javascript, C++ and so on. The Smart Contracts creation by these languages is easier than programming on other Smart Contract specific languages (eg. Solidity). On Sawtooth we have even a possibility to execute Ethereum Virtual Machine bytecodes that was compiled from Solidity code. Sawtooth even allows to connect to other type of blockchain as Ethereum and communicate with it. To resume Sawtooth is a modular, private blockchain that was a requirement of our partner Renault. In Section III we give information about the scalability of this blockchain on an IoT device, thus on a Raspberry Pi 3 B+ model.

III. IoT Model for Sawtooth Blockchain Applications

As we mentioned above we would like to use and adapt the BCM2837 architecture to our Sawtooth blockchain application. The Broadcom chip integrates a quad-core ARM Cortex-A53 and is used in Raspberry Pi 3.

A. Client application on Raspberry connected to the Validator node

Sawtooth is implemented in Python embedded in Docker images. These images are not compiled for the ARM processors that used in the major embedded architectures. That is why in this solution the Raspberry doesn't take part physically on the blockchain, it is only a client. To the best of our knowledge today there exists only one related work integrating an IoT network on Hyperledger Sawtooth [17]. This solution describe an Agri-Food supply chain management. IoT sensors send the captured data on a Hyperledger Sawtooth blockchain to achieve a full tractability of a life cycle of product (from production to selling). As in our case in this study the nodes sending data to the blockchain are off-chain nodes. These nodes communicate with the Validator via a REST API as in our work. Contrarily to our work in this solution the off-chain nodes are not IoT devices (PC's were used).

Because of the low maturity of Sawtooth C++ SDK we build our own program and library. This allows to format data and to send them to the Validator node via a REST API. Our program uses some special libraries as *protobuf* and *curl* because the REST API uses Google Protocol Buffers to make easier the communications and curl is used to send the data to the REST API by HTTP. To be able to send data to the Validator the program generates private-public key pairs using Secp256k1 elliptic curve. In addition to the key's generation, the payload is hashed by SHA512 cryptographic functions. The payload that we send are based on JSON format that will be handled later by a Transaction Processor (our Smart Contract).

B. Architecture adaptation

To adapt this architecture to our Sawtooth application we use the SystemC-TLM functional model of the BCM2837 that was deployed by the start-up Hiventive [18]. The modified architecture is shown in Fig. 4. It has to be noticed that the CPU is emulated with QEMU open source machine emulator and virtualizer. This architecture with QEMU supports a Debian Kernel (Linux version 4.19.0-2-arm64).

The basic architecture doesn't contain the SHA-256 and SHA-512 Intellectual Properties (IPs) that are hardware accelerators.

C. Analysis of our program

To be able to increase the performance and eventually decrease the energy consumption of our model we analyzed our program that is based on a determination of the most called functions during the program execution.

To better understand the dependencies between the function calls we use *gprof* GNU profiler [19] that can determine a Call Graph of the functions of a program. To be able to use this tool the program must be compiled with the option *-pg*. In addition to *gprof* we equally measured the elapsed time in the functions.

The analysis was made on a Raspberry Pi 3 B+ (with ARMv8 CPU). Scenario of the analysis:

Sending data to IntKey Transaction Processor:

- This is an default transaction processor of Sawtooth. The data that is sent is based on a JSON format, (a list of *key:value*). The data that we send sets/increments/decrements the value of a variable in the transaction processor. The size of the data is around 128 Bytes.

IV. Experimentations and Results

The 3 measurements below are more focused on our use case. We can imagine that a car will send more data than 128 Bytes (maybe images, sensors data and radars raw's data and so on):

- Sending 1 MByte data on dedicated car Transaction Processor.
- Sending 2 MBytes data on dedicated car Transaction Processor.
- Sending 41 MBytes data on dedicated car Transaction Processor.

A. Results of the program's analysis

In every scenarios the program uses *JSON* data format with *cbor* serialization (required for the transaction payload). To respect the format of the payload the program does conversions between *string, char* and *byte* types. Because of these conversions more than 85% of the total execution time was spent in functions related on these type conversions. It is probably possible to perform some optimization to reduce this number of call, possible only on software level modifications. In our study we rather look for a hardware level solution.

During the program execution there is a high number of cryptographic hash function calls, mainly SHA-256 and SHA-512. These functions could be computed rather on ASIC (Application-Specific Integrated Circuit) than on CPU. Using ASIC for solving a specific function is generally faster and consume less energy than CPUs.

1) Sending data to IntKey Transaction Processor: The total execution time takes 33.2 ms and the SHA-256 and SHA-512 take less then 0.02% of the total time. However the Elliptic Curve Digital Signature Algorithm (ECDSA) using Secp256k1 elliptic curve that was used for creating private-public key pairs takes 3.82% of the total time. It is hard to find full ASIC implementation of ECDSA using Secp256k1. However this procedure uses SHA-256 hash functions taking 17% of total time spent in ECDSA algorithm.

2) Sending 1 MBytes data on dedicated car Transaction Processor: The total execution time takes 4.495s. The payload of the message that sends the data is hashed. Because of this, SHA-512 is called 16573 times and it takes 3.46% of total execution time. The creation of one hash takes 9.382μs.

3) Sending 2 MBytes data on dedicated car Transaction Processor: The total execution time takes 8.76s. SHA-512 is called 32435 times and it takes 3.49% of total execution time. The creation of one hash takes 9.433μs.

4) Sending 41 MBytes data on dedicated car Transaction Processor: The total execution time takes 161.931s. SHA-512 is called 641047 times and it takes 4.04% of total execution time. The creation of one hash takes 10.21 μs.

The results of these 3 latest analysis is summarized in the Tab. I.

TABLE I
SUMMARIZED EXECUTION TIME (DATA $\geq$ 1 MBYTE) REPORTED AS AN AVERAGE OF 10 RUN

Data Size	Total exec. time	Time occupation of total time by SHA-512	Time of creation of one SHA-512 Hash
1 MByte	4.495 s	3.46 %	9.382 μs
2 MBytes	8.76 s	3.49 %	9.433 μs
41 MBytes	161.931 s	4.04 %	10.21 μs
Average :		3.66 %	9.675 μs

The Tab. II summarizes execution time when data is less than 1 MBytes.

TABLE II
SUMMARIZED EXECUTION TIME (DATA < 1 MBYTE), WHERE ECDSA $\rightarrow$ SHA-256 MEANS THAT SHA-256 TAKES 17% OF THE EXECUTION TIME OF THE TOTAL TIME EXECUTION OF ECDSA

Data Size	Total exec. time	Time occupation of total time by SHA-512 & SHA-256	Time occupation of total time by ECDSA $\rightarrow$ SHA-256
< 1 MByte	32 ms	0.02 %	3.82 % $\rightarrow$ 17 %

We can notice that when data type is greater then 1 MBytes, SHA-512 functions takes 3.5-4% of the total time and SHA-256 becomes negligible when the size of data is big.

B. BCM2837 modified functional SystemC model

After the results of the program's analysis we found that with hardware accelerators realizing the cryptographic hash functions as SHA-256 and SHA-512 we can obtain a significant gain. In this modified BCM2837 model we added this

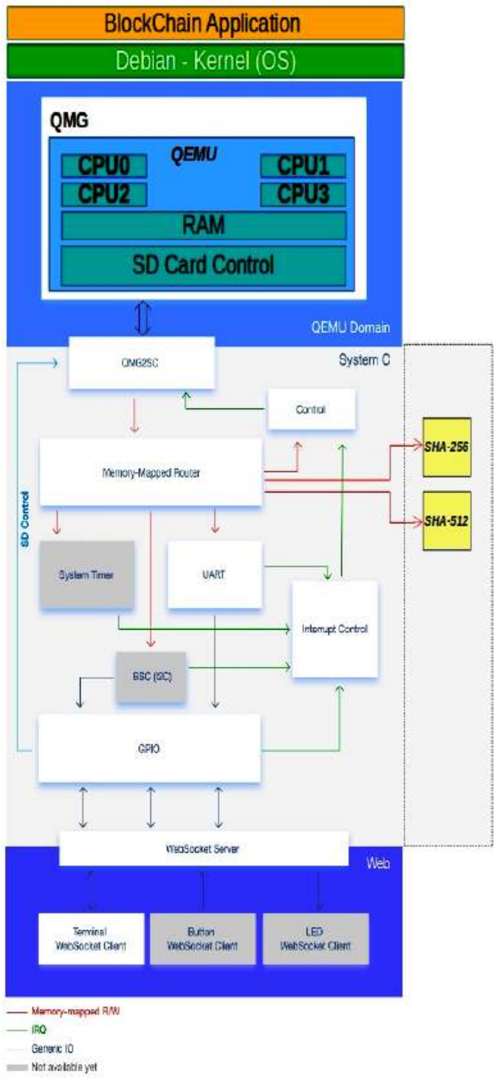

Fig. 4. BCM2837 model with hardware accelerator IPs

type of hardware accelerator modules shown on Fig. 4. The IPs are connected to the Memory-Mapped Rooter that is equal to a traditional bus.

As we mentioned earlier on the BCM2837 we are able to run a Debian Kernel thanks to QEMU. When the Kernel is in the boot process it needs Device Tree Blob (DTB) containing information about the hardware components that the architecture has. These information are the addresses and the partitions of these components. The Kernel has to be informed about the new IPs (SHA-256 and SHA-512) that we add to the architecture, thus we have to indicate the addresses that will be used by these IPs. We add addresses to the Device Tree Source (DTS) that will be compiled into a DTB format.

To get access to the IPs from the user application (from our program) we deployed Kernel Drivers for it. From the user application we don't have direct access to the IPs, because their addresses are accessible only by the Kernel. In our program the SHA-512 function is called from Crypto++ (Version 8.3) cryptographic C++ standard library. The SHA-256 function in ECDSA algorithm was implemented in libsecp256k1 C++ library. In these libraries we modified the functions SHA-512 and SHA-256, when these functions are called form the user application they call the dedicated Kernel Drivers that interact with the associated IP.

TABLE III
PROCESS TIME AND GAIN, THE ASIC ESTIMATED FOR 40 NM CMOS TECHNOLOGY

	Process time CPU	Process time ASIC	Gain
SHA-512	9832 ns	32 ns	**293.18**
SHA-256	2760 ns	24.48 ns	**112.75**

To be able to give an idea of the gain that we can achieve with hardware accelerators we were strongly inspired by the analysis of SHA implementations of the University of California and Katholieke Universiteit Leuven [20]. In this study the SHA functions were implemented on ASIC based on 130 nm CMOS (Complementary Metal Oxide Semiconductor) transistor technology. The Raspberry Pi 3 B+ models are realized on 40 nm CMOS technology. By using lower size CMOS technology of CMOS the delay will also lower. To estimate the delay for that 40 nm technology we used the study of the scaling equations for the accurate prediction of CMOS device performance for 180 nm to 7 nm [21].

After the delay estimation for the 40 nm CMOS technology we compare the execution time of one hash of SHA-256 and SHA-512 executed on the Raspberry (on it's CPU) and executed on the ASIC hardware accelerators. These execution times of only one hash creation and the gain that can be obtained are summarized in the Tab. III. We have to notice also that to reach these gains the SHA hardware accelerators must work at a frequency of 746MHz for the SHA-512 and 794MHz for the SHA-256.

When the data size that will be sent is greater than 1 MByte without the hardware accelerators we saw that SHA-512 takes 4-3.5% of the total execution of the program. By using SHA-512 IP we can decrease this usage to 0.012%. In the case when the data size is less then 1 MByte in ECDSA Secp256k1 algorithm the SHA-256 function was used 17% of the total time of ECDSA algorithm. The hardware accelerator allows to decrease this percentage to 0.14%.

C. Current Consumption of Raspberry Pi 3 B+ during the application executions

In this part of our work we measure the current consumption of the program execution according to the data size that is sent. The results are given with and without the Idle mode consumption of the Raspberry. The Tab. IV represents these results. The results show that the current consumption is quasi linear in terms of the data size.

We notice that the execution of the OS on Raspberry Pi consumes around 0.52 A. The Fig. 5 represents the curves

TABLE IV
CURRENT CONSUMPTION WITH AND WITHOUT OS ACCORDING TO THE DATA SIZE TO SEND

Data size to send	Current consumption with Idle	Current consumption without Idle
$<$ 1 MByte	0.13 A.s / 0.0365 mAh	0.024 A.s / 0.0065 mA.h
1 MByte	3.37 A.s / 0.94 mA.h	0.76 A.s / 0.21 mA.h
2 MBytes	6.29 A.s / 1.75 mA.h	1.51 A.s / 0.42 mA.h
41 MBytes	153.07 A.s / 42.52 mA.h	24.06 A.s / 6.67 mA.h

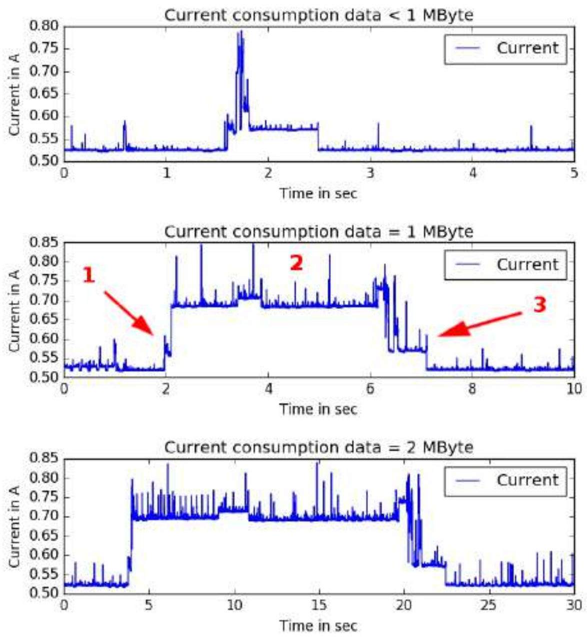

Fig. 5. Current consumption on Raspberry Pi

of the current consumption. This figure doesn't represent the irrelevant case of 41 MBytes because its execution time is around 40 times higher than in the case of 1 MByte data. We can also notice in the curves that the more data is sent the more is the power consumption, because the C++ application has more computation to do. We can equally see the different phases of the current consumption. The program is started in the first phase, is executed in the second phase, and the memory is freed in the third phase.

V. CONCLUSION

In this paper we studied the implementation of Hyperledger Sawtooth on a Raspberry Pi 3 B+. A full node of Sawtooth is not already implementable on the Raspberry because of the dependencies of Docker images that are not compatible with ARMv8 processor used by the Raspberry. We figured out with an alternative solution where the Raspberry can connect to a node of Sawtooth but the node is deployed on a required architecture (PC X86_64). For this solution we deployed our own client C++ SDK for Sawtooth.

In the second phase of our work we studied the most called functions by our application. We modified the BCM2837 architecture by adding hardware accelerators computing SHA crytpographic functions. The results show that we can obtain a high gain of the performance. We equally measured the current consumption of the program on the Raspberry.

In latest studies we would like to determine a power controlled architecture model of the BCM2837. By the power controlled model we would be able to determine a power management to decrease the power consumption of the program and equally the whole architecture.

In more advanced phase of our research we could modify our model running the use case application on other types of blockchain as Ethereum and IOTA, but still achieving a high performance and optimal energy consumption.

ACKNOWLEDGMENT

This work has been supported by the French government, through the UCA^{JEDI} and EUR DS4H Investments in the Future projects managed by the National Research Agency (ANR) with the reference number ANR-15-IDEX-0001 and ANR-17-EURE-0004.

REFERENCES

[1] S. Madakam, R. Ramaswamy, and S. Tripathi, “Internet of Things (IoT): A literature review,” *Journal of Computer and Communications*, vol. 3, no. 05, p. 164, 2015.
[2] Accessed: 2018-10-14. [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide
[3] G. Arnout, “SystemC standard,” in *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, Jan 2000, pp. 573–577.
[4] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery, “Sawtooth: An Introduction,” *The Linux Foundation, Jan*, 2018.
[5] K. Christidis and M. Devetsikiotis, “Blockchains and Smart Contracts for the Internet of Things,” *IEEE Access*, vol. 4, pp. 2292–2303, 2016.
[6] M. Conoscenti, A. Vetr, and J. C. De Martin, “Blockchain for the Internet of Things: A systematic literature review,” in *2016 IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*, Nov 2016, pp. 1–6.
[7] K. Biswas and V. Muthukkumarasamy, “Securing smart cities using blockchain technology,” in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Dec 2016, pp. 1392–1393.
[8] F. Verdier, P. De Filippi, T. Marteu, F. Mallet, P. Collet, L. Arena, A. Attour, M. Ballatore, M. Chessa, A. Festré, and P. Guitton-Ouhamou, “Smart IoT for Mobility: Automating of Mobility Value Chain through the Adoption of Smart Contracts within IoT Platforms,” in *17th Driving Simulation & Virtual Reality Conference (DSC 2018)*, 2018.
[9] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, “An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends,” in *2017 IEEE International Congress on Big Data (BigData Congress)*, June 2017, pp. 557–564.
[10] N. Szabo, “Smart contracts: building blocks for digital markets,” *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, 1996.
[11] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.
[12] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, “Efficient Byzantine Fault-Tolerance,” *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, Jan 2013.

[13] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: ACM, 2018, pp. 30:1–30:15. [Online]. Available: http://doi.acm.org/10.1145/3190508.3190538

[14] Accessed: 2019-06-13. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/software-guard-extensions.html

[15] Accessed: 2019-06-14. [Online]. Available: https://sawtooth.hyperledger.org/docs/core/releases/latest/contents.html

[16] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi, "On Security Analysis of Proof-of-Elapsed-Time (PoET)," in *Stabilization, Safety, and Security of Distributed Systems*, P. Spirakis and P. Tsigas, Eds. Cham: Springer International Publishing, 2017, pp. 282–297.

[17] M. P. Caro, M. S. Ali, M. Vecchio, and R. Giaffreda, "Blockchain-based traceability in agri-food supply chain management: A practical implementation," in *2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany)*, May 2018, pp. 1–4.

[18] https://www.hiventive.com/, Accessed: 2019-07-18.

[19] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A Call Graph Execution Profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, Jun. 1982. [Online]. Available: http://doi.acm.org/10.1145/872726.806987

[20] Y. K. Lee, H. Chan, and I. Verbauwhede, "Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations," in *Information Security Applications*, S. Kim, M. Yung, and H.-W. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 102–114.

[21] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180nm to 7nm," *Integration*, vol. 58, pp. 74 – 81, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167926017300755