Modern Management based on Big Data III
A.J. Tallón-Ballesteros (Ed.)

doi:10.3233/FAIA220113

A Multiprocessing Framework for Heterogeneous Biomedical Embedded Systems with the Proposal of a Finite State Machine-Based Architecture

Xiaohe TIAN[1] and Mang I VAI
Faculty of Science, University of Macau, Macao, China

Abstract. The applications of heterogeneous embedded systems for biomedical engineering are promising, as quick response of biomedical systems is often required due to the life-saving nature of biomedical engineering, and multiple devices with completely different cores and designs can be involved in a single patient. In this paper, we propose a multiprocessing framework and then, with regard to the framework, we propose an architecture for heterogeneous embedded systems that uses finite state machines (FSMs). A multithreading method on an electrocardiogram (ECG) software is implemented as the verification of our framework.

Keywords. Multiprocessing framework, heterogeneous embedded systems, finite state machine.

1. Introduction

Embedded systems play an important role in biomedical systems due to their response speed, reliability, portability, and ability to be designed for a specific biomedical use. Meanwhile, heterogeneous multi-core embedded systems take advantage of not only their potential in reducing power consumption, but the variety of techniques and cores used by biomedical devices for different patients and diagnoses. Therefore, we consider these two problems of (1) how embedded systems should be configured for specific biomedical uses, and (2) how we can improve the efficiency of the systems.

At the side of biomedical system configuration, T. Hussain et al. developed a Biomedical Application Processing System which made use of both dual-core processor and Biomedical Application Specific Reconfigurable Accelerator (BASRA) [1]. Then, at the side of system efficiency, G. Xie et al. proposed models for hardware cost, reliability requirement (RR) and real-time requirement assessments, and then introduced exploratory hardware cost optimization (EHCO) algorithm with its derived algorithms, EEHCO and SEEHCO [2]. While those works give specific ideas about efficiency improvement and biomedical applications potentially feasible for embedded systems, we try to incorporate those ideas, among others, by introducing a

[1] Corresponding author, Department of Electrical and Computer Engineering, University of Macau, E11, Avenida da Universidade, Taipa, Macau, China; E-mail: mb85468@um.edu.mo.

multiprocessing framework and then, with regard to the framework, proposing an architecture for heterogeneous embedded systems that uses finite state machines (FSMs). The framework will be introduced in Section II, while the architecture proposal will be introduced in Section III. Then, as a verification to the framework and architecture, a multithreading method on an electrocardiogram (ECG) software is implemented in Section IV. We conclude in Section V.

2. Multiprocessing Framework

We propose a multiprocessing framework as this: for an application which executes every input through a fixed set of tasks, we divide it into processing units of finite number. The tasks assigned to each of the units are also fixed, so that they can form a task dependency graph (TDG) that is a directed acyclic graph (DAG). Examples of TDGs, one DAG and the other non-DAG, are shown in Figure 1.

We propose this framework with considerations on (1) efficiency, so that time wasted through task switching can be reduced and memory access schemes can be simplified, with every processing unit administrating a fixed set of tasks; (2) simplicity and flexibility, so that manual segmentations of tasks are possible in the absence of automatic task segmentation schemes, with the number of processing units being finite; (3) availability for real-time applications.

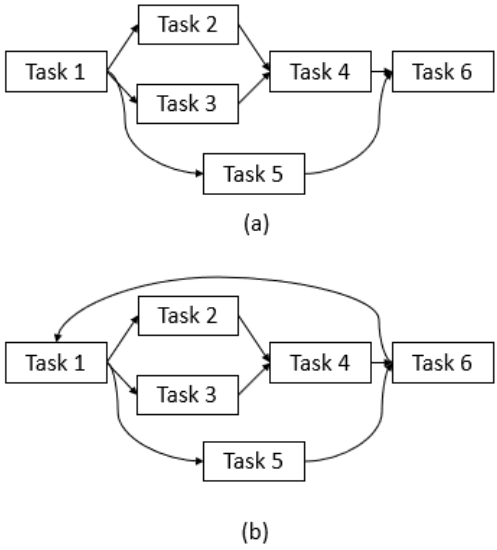

Figure 1. Examples of task dependency graphs (TDGs) from the perspective of a single input. Here, every box represents a set of tasks enclosed in a processing unit, and every edge represents a dependency between two sets of tasks. In (a), no edges form cycles, and hence it is a directed acyclic graph (DAG), while for (b), because of the extra dependency from Task 6 to Task 1, Tasks 1-2-4-6-1 form a loop, and the graph is not a DAG.

If the TDG is a DAG, all processing units can execute tasks in parallel, and both multiprocessing schemes, namely concurrent processing and parallel processing, can be used in accordance with the TDG. However, in case the processing units are not already well-defined or the dependencies of tasks in them cannot form a DAG, task

segmentation schemes need to be proposed. The difficulty comes at making the TDG a DAG. As illustrated in Figure 1 (b), any cycle formed in a TDG will make parallel processing unavailable. Makeshift methods, such as altering the algorithm and using extra memory for breaking the cycles of task dependencies, can be applied for making up DAGs, while efficient and automatic methods for breaking the cycles of dependencies are subject to further research.

3. Embedded System Architectures

With our multiprocessing framework, we propose an architecture for embedded systems, aiming at taking advantage of the framework. An overview of the proposed architecture is given in Figure 2. As it shows, every processor has a finite state machine (FSM) attached to it, and a memory space for data access and storage. There is also a simple data processing unit (DPU), which contains concurrent processor algorithms, a memory supervisor as a centralized controller for the memory spaces of processors, and the necessary memory space for DPU itself. FSMs communicate with other components by sending and receiving signals. Here in this figure, reading request (R_r) and writing request (W_r) signals that FSMs send to DPU, as well as reading finished (R_f) and writing finished (W_f) signals that DPU sends to FSMs, are presented.

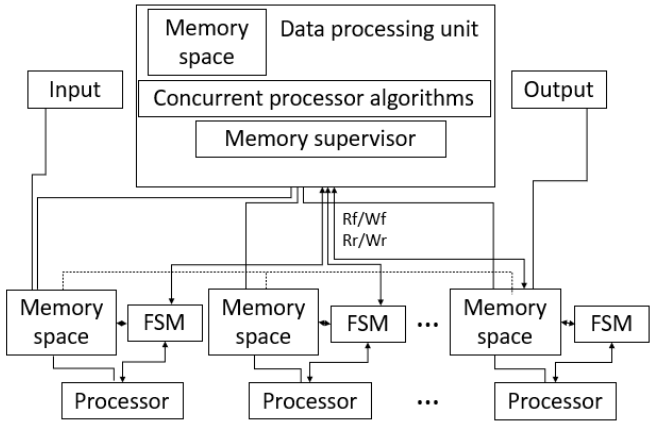

Figure 2. Proposed embedded system architecture based on our framework. Unarrowed edges indicate the communications between those components are by data passages or supervisions. Arrowed edges indicate the communications are by transitory signals. Dotted edges indicate the communications may exist but are dependent on other conditions.

Figure 3 (a) and (b) show our proposed FSMs, namely FSM1 and FSM2. FSM1 is used for all processors except terminal processor(s), which are processors with no "next processors", i.e., no processors dependent on them. An example is Task 5 in Figure 1 (a). Terminal processors use FSM2 because they do not need a waiting state.

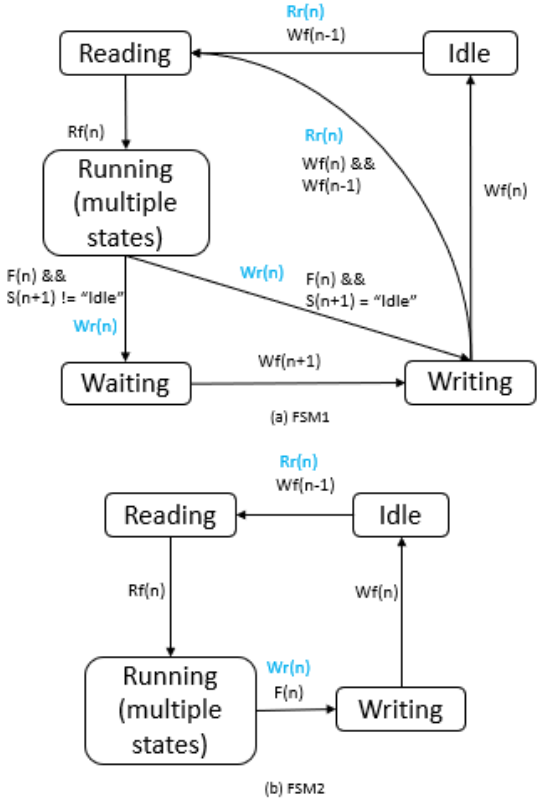

Figure 3. Two FSMs, namely (a) FSM1 and (b) FSM2, used according to our proposal. F indicates the task finishing signal sent from the respective processor. S indicates the state of the FSM of the respective processor. Inside the bracket, n indicates the processor the FSM corresponds to (current processor), and n-1 and n+1 indicate the last and next processors. Outputs with the respective edges are colored in blue. "&&" is logical "and" and "!=" is "not equal to".

Concurrent processor algorithms generate collective W_f signals, or W_f^c in case there are more than one last/next processors for a specific processor. We apply a counter in order to indicate the generation of the W_f^c for every case where the number of last/next processors is larger than 1. Starting from 0, the counted number is added by 1 every time an individual W_f is generated, and once it matches the quantity of last/next processors in the respective case, the W_f^c is generated, and then the number returns to 0. Take Figure 1 (a) for example, W_f^c should be generated for Task 1 as $W_f(n+1)$ and for Task 4 as $W_f(n-1)$. There is a counter for each of the cases, and both has their W_f^c generated when the number reaches 2.

The architecture should be able to take the advantage of simplicity in terms of data storage. According to Figure 3 (a), for example, since there are 5 major states for non-terminal processors, 3 bits of memory spared for every FSM of the first type is enough for the key part. For terminal processors, it will be 2 bits. As for the multiple states in the running part, since the operations are serial and non-stop, primary memory comes into consideration along with secondary memory. For example, if a byte of secondary memory is arranged for a non-terminal FSM, there will be 5 bits for the states in

running part. Then, primary memory will be used if the number of operations exceeds $2^5 = 32$. The number could also be less than 32 so that more primary memory is used to improve the speed.

As for data access, since the advantage of inter-processor data passages is the usage of a fixed memory space for every processor, the process of data passage can be largely automated, resulting in distributed architectures. Nonetheless, we need a memory supervisor to decide when to enable data passage between the memory spaces of two processors with dependency, regarding the W_r or R_r signals it receives. After the data passage is finished, the memory supervisor sends a W_f signal to the FSM of the writing processor and R_f signal to that of the reading processor. We can skip the issue of priority by applying an individual bus to every FSM for sending W_f signals.

We divide the conditions for data passages into two types: 1) uncrowded, which means when the writing processor gets its data ready, the FSM of the reading processor is at idle state, so the data passage can take place immediately; 2) crowded, which means when the writing processor gets its data ready, the reading processor is not at idle state, so the writing processor has to wait, and the data transition has to take place later. Because a processor stops reading new data and has its data stored in its memory space when its FSM is at waiting state, it owns only one set of data at one time, and the data will not stack up infinitely. And because the terminal processor(s) do not have waiting states, the non-terminal processors will not have its FSM at waiting state indefinitely either. In case the processors with their FSMs in waiting states stack up, and the enabled memory spaces form a chain in a TDG, we call it a waiting chain. This concept is potentially useful in improving the efficiency of the architecture, as when the end processor(s) in a waiting chain finished running tasks and writing data, the data passage on the chain can happen as quickly as possible. Thus, efficiency improvement schemes, such as using caches, can be investigated.

4. Test on Software

We tested the feasibility of our framework on software by modifying an open-source electrocardiogram (ECG) detection and classification program. [3] The objective of the test is to verify the availability of the proposed multiprocessing framework via the multithreading implementation, so maintained performances in accuracy and speed are primarily expected. If there are available resources in hardware such as thread executions being distributed to multiple cores, improved speed will also be expected.

The ECG program is written in C language, which is the same language to be used in our modification. The database it uses for performance evaluation is MIT-BIH database. [3][4] 25 ECG records are used in total, each of which has around 1000 to 3000 beats (QRS component) for detection and classification. Through code analysis, we divided the program into subtasks and determined their dependencies. Thus, they made up 6 modules as shown in the DAG of Figure 4, and their functionalities are shown in Table 1. Then, for multithreading implementation, we made use of the POSIX Threads model, or “pthread” header file. Every module was therefore fit into a thread. In code modification, we introduced a flag between every two individual threads where dependency exists. Writing and reading threads here correspond to writing and reading processors in our architecture. Collective flags, corresponding to collective W_f^c signals, are simply implemented by logical AND operators. For data

storage and access, we largely replaced variables by arrays, and arrays by double arrays, in order to store the buffering data to be used. Public data to be used by one thread were converted into private ones as an implementation of reading process. Similarly, private data were copied to public ones as a writing process. Proper mutual exclusion functions, such as "pthread_mutex_lock", were applied.

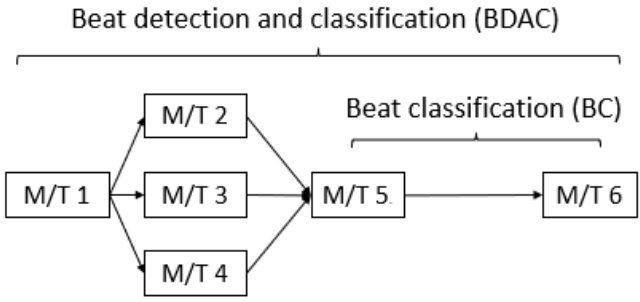

Figure 4. TDGs with our task segmentation method for testing. M/T represents module/thread.

Table 1. The functionality within every module

No. of module	Main functionality
1	Read sample, init, detect beat, etc.
2	Estimate noise
3	Downsample beat
4	Update beat queue
5	BC init, check muscle noise, check rhythm, analyze beat, etc.
6	BestMorphMatch (find the template that best matches the beat), update beat type, other BC tasks, other BDAC tasks, output beat type, etc.

We tested our modified program using GNU Compiler Collection (GCC) for compiling and compared it with the original program. The computer we used for testing is installed with the Intel i7-6700 CPU which has 4 cores, and Microsoft Windows 7 operating system. We tested the running time for every record using (1) the "clock" function via "time.h" header, and (2) a manual timer, according to when the start of execution of every record was printed. With the clock function method, the printed running time per record for the original program is around 500 – 1,000 ms, while for our multithreaded method it is around 90,000 – 120,000 ms. According to the manual timer, however, the running time per record is around 14 - 17 s for both programs. In terms of accuracy, we achieved "almost" the same performance in accuracy as the original program for the multithreaded program. Figure 5 shows some of the results to illustrate our observations. The results from manual timer method meet our expectations and are more convincing to us as the time measurement inside the program may be disrupted by pthread functions. As for the fact that no speed improvements are observed, as is understood, there are several obstacles in the effective time reduction for a multithreaded program such as the inability for user-level threads to save the running time, and the incompatibility between C language and multithreading functions as well as that between pthread functions and Windows OS.

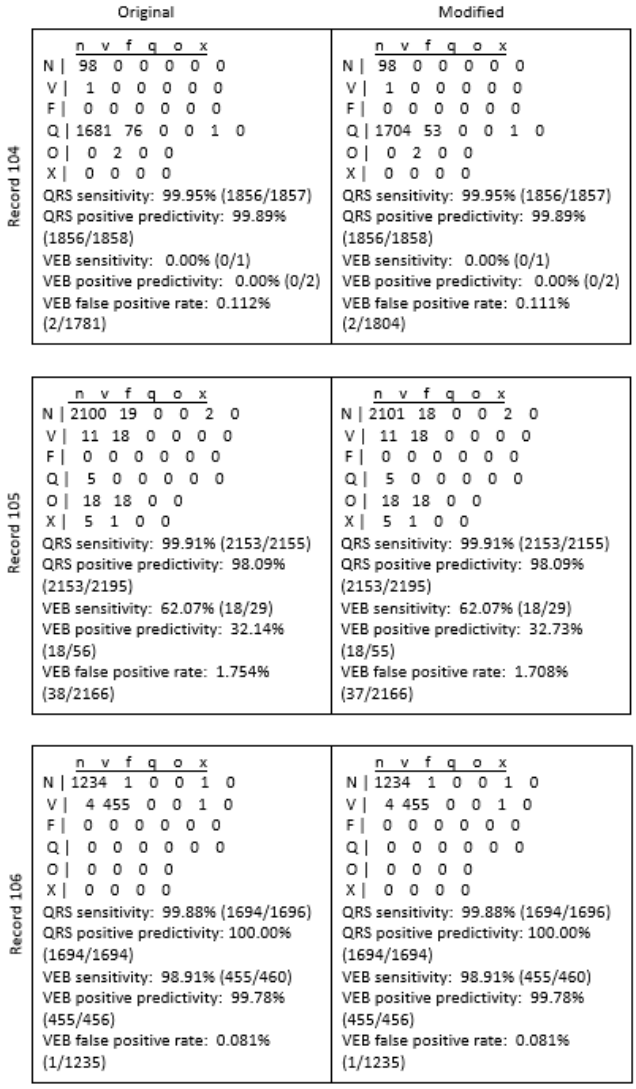

Figure 5. Test results of our modified multithreaded program for three ECG records (Records 104, 105 and 106) in comparison with the original results. Lowercase letters are beat types through classification, while uppercase letters are real beat types.

5. Conclusion

We proposed a framework for multiprocessing adjustable for biomedical applications, and then an embedded system architecture with finite state machines and a data processing unit that handles memory access and storage for different processors. We verified the feasibility of the framework by implementing a respective multithreading scheme on an ECG detection and classification program on software. In the future, we aim at addressing other issues such as load balancing and real-time system

architectures. as well as investigating in more specific biomedical systems and modifying our proposals accordingly.

Acknowledgements

This work was supported in part by the Shenzhen-Hong Kong-Macau S&T Program (Category C) of SZSTI (SGDX20201103094002009) and in part by the University of Macau (File no. MYRG2020-00098-FST).

References

[1] Hussain T., Haider A., Taleb-Ahmed A., "A heterogeneous multi-core based biomedical application processing system and programming toolkit". Journal of Signal Processing Systems, vol. 91, no. 8, pp. 963-978, 2019.
[2] “Hardware Cost Design Optimization for Functional Safety-Critical Parallel Applications on Heterogeneous Distributed Embedded Systems”
[3] Hamilton P. (2002), "Open source ECG analysis," Computers in Cardiology, 2002, pp. 101-104, doi: 10.1109/CIC.2002.1166717.
[4] The MIT-BIH Arrhythmia Database CD-ROM. Available from the Harvard-MIT Division of Health Sciences and Technology, 1992.