Embedded Development Being Future for Software Engineering

Herbert Yeung, Steve Murray

Faculty of Engineering
University of Technology Sydney

Abstract

This paper is aimed at software engineering students and curriculum advisors for the software engineering degree/course, as it delves into the emergence of embedded development being the future direction of software engineering.

It is important to consider engineering from different perspectives, as software engineering is an adaptable discipline due to the abstract nature of software. The definition of software engineering is often vague as it can be applied to a wide range of areas and fields [1]. This paper will consider hardware in software engineering, as the usual abstraction between software and hardware is slowly eroding, especially in embedded development.

The research conducted has been motivated primarily by three factors. The first is the capabilities of using embedded devices in development, followed by the availability of open source tools aiding engineering students in achieving this. Furthermore another key motivation is to demonstrate that hardware issues need to be considered, as there are limited resources when developing under an embedded environment. The importance of considering parallelism when developing is also stressed, as this is often neglected when software students design their systems/solution.

This research is significant because it demonstrates that students cannot just abstract themselves and be concerned only with software design and development aspects; importance must be placed on other aspects as well, such as hardware and mechanical issues.

1. Introduction

Part of the software engineering doctrine has been a slow dissemination between hardware and software. One of the underlying reasons for this achievement in abstraction is the choice of operating systems, such as Linux and other such variants [7]. This has guided the acceleration in software development as the use of existing tools that come with an operating system has helped aid in the efficiency of development. However, the tools do not mean that the software engineer needs to be void of the hardware issues that still exist when developing. The choice that face most application developers is to develop for graphical user interfaces [2]. However, as resources are usually more restricted for embedded development, the future challenge may lie in other forms of development, such as sensors, as they are usually integrated into embedded systems to make them applicable.

Another area that this prototyping delves into is the consideration of parallelism. It is an issue that is often not considered during the design stages of development due to applications tending to be produced using a single processor in mind [6]. This means that design concerns such as concurrency issues or the ability to identify performance bottlenecks are often ignored. Consideration of parallelism will help shed light on complex problem for the

software engineer, as there are more design options that are available, including the use of parallelized algorithms and processors as described by Wilkinson, B. and Allen, M. [6].

2. Development Requirements

A requirement of this investigation is to consider the use of operating systems to provide foundations for parallel processing and the investigation of parallel processing initiatives. Programming paradigms and changes to programming methodology when developing a parallel application often do not consider the implications it has while operating under an embedded environment. This research aims to cover the issues that are dealt with under embedded development.

3. Design of System

Conceptually, the design of the project is for an embedded device that has sensors to mimic real world usage. These sensors will generate output signals to the board to show user interaction. The sensors are to also receive input signals to indicate any changes to the system, such as the invocation of a process.

The architecture of the application is of a centralized network, whereby the central node distributes the known parallelized code to any device that is waiting to receive the executable. Each node must have the required ability to retrieve the executable; this means that a communication link must be established. The main purpose of this design is to allow the storage and the distribution of the application to occur on the server side [3]. This is illustrated in diagram 1.

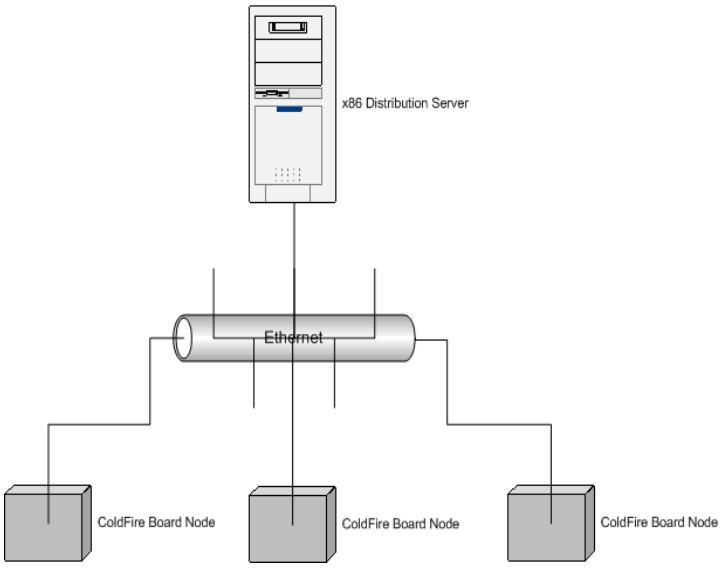

Diagram 1: Architecture of the system

The choice of hardware was an important part of research. Most non-embedded hardware development requires some form of memory management. Thus, a hardware platform was chosen that gives more challenge to the undergraduate software engineer. uClinux is an

appropriate embedded operating system that has primitive memory management. Chipsets that could run uClinux included the Motorola® Coldfire™ processor.

The first stages of the research involved compiling a set of preexisting tools to help with the software development. Linux and other such variants provided a plethora of open source tools. For development it was decided upon to use Flex and Bison with the aid of the Gnu C Compiler (gcc). The choice to use these tools was based on the goal of demonstrating the identification of processing bottlenecks. Flex was used to preprocess the code to be parallelized and stored the results. The stored results were then copied into templates that provided socket communication. Gcc was then used to cross compile the source code for the Coldfire™ board and the x86 machine.

For the second part of the prototyping, consideration needed to be made regarding the hardware interfacing of the development board. The Sentec® Coldfire™ development board had a major advantage being that peripheral ports that contained digital I/O capabilities. This made it easier for the developer to add devices to the board. Diagram 2 represents this in action. Upon this knowledge, an interface was created to interact with light emitting diodes and a light variance detector.

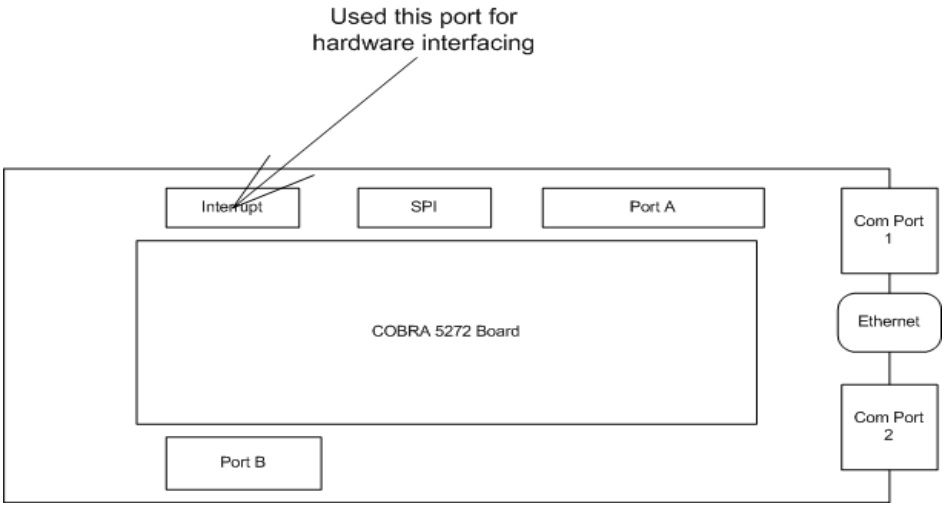

Diagram 2: Demonstration of the available ports on the board. (Adapted from the documentation provided by Sentec® [4])

On the embedded platform, a daemon (background process) that is running awaits an interrupt from the user to indicate the node is idle. The interrupt generated is from a hardware device attached to the interrupt port of the Sentec® board. This invokes the embedded platform to load the distributed processing task to be executed. After completion of the task it returns the calculation value back. It then reloads the original application that the device was originally running and runs it until the next invocation.

The application works by extracting code that can be parallelized before compilation. Based on this, separate executables are created depending on the number of processor nodes attached on the system. Each executable is compiled based on the processor architecture that it is targeted for. The separate executables are then stored on the server for later retrieval by a

node when the node is in an idle state. These executables will start running upon invocation from the daemon (background process). Once the executables have completed their assigned processing tasks, the results are sent back, using sockets, to the server which will collate the data and merge it to set the final answer.

The design of the hardware showed many aspects that needed to be considered. The maximum and minimum voltage input/output of the device and the current were needed to determine how the. Choice of port selection was also important, as the system needed to be of an interrupt sensor than rather a polling sensor to simulate asynchronous events. This meant that research needed to be conducted on the right pin selection within the port to allow for proper timing to be registered.

A major limitation of this design is that if the user wants to use the embedded device, mimicked by a secondary interrupt, any processing currently being performed will be lost. Future enhancements will need to be made to render this limitation.

4. Results

The use of an operating system for an embedded platform allowed greater ease in accessing hardware components while at the same time it also showed the limitations that were involved in developing under an embedded environment. One of the restrictions comes from the limited amount of memory and stack size for a process.

An application using a complex 'for loop' algorithm of computation showed that on a normal system, the processing time taken for development varied. However, processor utilization was maximized. This is significant as it shows that embedded systems have the ability to be maximized as opposed to idling, which is what most real world embedded devices do.

The results were derived from measurement of the process' termination time conducted to determine the performance of the system. Linux provides API calls that allow the developer to measure the time it takes for a given process to complete.

The results also showed that under single processor architecture the CPU utilization was maximised. The extent of this CPU utilization did not equate to greater efficiency for processing for most of the programming tasks assigned. For large programming tasks, if required hours of processing time was uninterrupted, and the embedded device had no network failures, showed slight improvements in performance. The bottlenecks that were minimized using parallel processing with embedded devices attached on a heterogeneous network included complex 'for loops' in C programming code.

Based on the current research, the values obtained indicate that on an application that did not require processing the application did not have any performance gain. When the complexity of the code rose the time taken was also. Initial readings show that there was only a slight performance gain of about 5.4%. This could be due to a number of limitations that were not considered in the design.

Task	Time Without Distributed Processing	Time With Distributed Processing
100 Iterations	3615.25s	3615.25s
1000 Iterations	18720.33s	17784.31s
10000 Iterations	61238.28s	57528.98s

Table 1: Some results conducted from the research

Limitations were also found in performing tests and measurements on the performance of the application. Some of these limitations included measuring network latency, which hindered the transfer of data. Another limitation involved the collation and merging time taken by the central process running on the server. There were also limitations in properly identifying other avenues of parallelism. This failure shows that the program currently is restricted in its performance and will need further enhancements. Some possible additions include the better identification of parallelized code; better load balancing and the saving of already processed data so that it could be resumed than rather restarted.

Results indicate parallelism can be achieved by integrating certain embedded devices into a heterogeneous network, though for small applications significance in processing efficiency is negligible.

5. The Need to Consider Parallelism in Development

Most software is developed without parallelism in mind, leading to many situations where a lack of resource utilization exists [6]. The undergraduate software engineer could attribute this to the lack of design decisions. However, as most other paradigms and protocols have identified, the adoption of parallelism is sometimes complex and there exists a strong reliance on the software engineer to manually identify points of parallelizable code.

Measurements on the network latency and the processor utilization show that the availability of tools to help perform these measurements aids in the testing phase of the software development under an embedded environment. This is beneficial to the undergraduate software engineer because exposure to these tools would aid with the efficiency of development.

The integration of sensors to a preexisting embedded device allowed insight into the design of hardware. It also indicated that a set of requirements considered hardware performance as bounded not to just the input/output of the processor, but also the input and output provided by the peripheral devices.

6. Advanced Operating System Curriculum

Adoption of real-world parallel development would have implications of a more advanced look into operating systems as part of the software engineering curriculum. This subject will aim at emphasizing the importance of considering parallelism and the use of tools will help achieve this. It will also look into embedded development and consider the use of an embedded device to use for real world applications.

Experiments using tools and operating systems like Amoeba [5] on hardware platforms like the Coldfire™ cluster will permit a select group of students to look at distributed embedded platforms and the issues that are particular to architectures like this. These could be evaluated at varying levels of abstraction from the hardware and provide opportunities to examine application areas like sensor networks.

7. Conclusion

Onus is on the software engineer to be not constrained to believe that this leaves him/her to only consider certain aspects of software. Rather software engineers need to be able to formulate solutions for hardware type problems or complexities as well as understand the importance of parallelism. This paradigm shift will help make the software engineer professional come up with better design solutions with the added knowledge and experience on hardware design and parallelization of software.

The use of software process is still paramount for the achievement of any development. However, the argument of this paper is to show that the software process needs to take into consideration of other issues, such as hardware and distributed computing. When it comes to embedded device development, this is what is required of the software engineer. Issues such as de-bouncing of switches; filtering of digital I/O; and parallelizing code will need to be considered and dealt with.

Further research could be undertaken to look into the adoption of this with hard real time requirements, as this will aid in accessing the viability of a more critical system that would be more applicable to real world usage. Most embedded applications are standalone and usually require no server and client communication. This research suggests that this will not be the case in the future as parallelism will be an integral part of most development.

References

[1] WikiPedia: Free Encyclopedia (2005)'Software Engineering' [Online] Available: http://en.wikipedia.org/wiki/Software_engineering, [2005, April, 25th]

[2] Brown, Chappell "GUI development system targets net appliances" Electronic Engineering Times, Issue 1106, p85, 27th March 2000

[3] Vrenios, Alex (2002) *LinuxÒ Cluster Architecture*, United States of America: SAMS Publising

[4] Sentec Inc. (2005)'COBRA5272 Embedded Processor Module: Carrier Board for the COBRA5272 Processor Module' [Online] Available: http://www.sentec-elektronik.de/COBRA5272_Product_Information.pdf [2005, April 20th]

[5] Tanenbaum, A.S., Kaashoek, M.F., Renesse, R. van, and Bal, H.: "The Amoeba Distributed Operating System-A Status Report," Computer Communications, vol. 14, pp. 324-335, July/August 1991.

[6] Wilkinson, B. and Allen, M. (1999) *Parallel Programming: Techniques and applications Using Networking Workstations and Parallel Computers*, United States of America: Prentice-Hall Inc., pp3-5

[7] Sevenich, R. (2003) 'Retraining for Embedded Linux Development' [Online] Available: http://linuxdevices.com/articles/AT8223538805.html [2005, April 20th]