Porting of Automotive Application on to Embedded Multicore Platform

Jijinraj P

Abstract— In modern computer hardware, increase in processing capacity is gained by introducing more processor cores but not with increased clock speed. This may imply reduction in processing capabilities as far as the individual cores are concerned. So it is necessary to partially restructure the applications that are not designed to benefit from multiple cores for maintaining the existing performance. To improve the quality of experience for drivers as well as passengers, multi-core embedded systems are getting adopted on a very large scale in automotive industry. But measures and methodologies for porting applications are yet to be standardized. Solving the issues related to it will open up interesting avenues for utilization of multi-core systems in automotive industry. This work emphasizes on porting legacy automotive application onto embedded multicore platform for better performance.

Index Terms— Embedded multicore, Loop Parallelization, cross compiler tool chain, OpenMP, Optimization, Automotive Applications

1 Introduction

Demand for higher performance in electronics was generally met by shrinking silicon chip and increasing the frequency of operation, but it was found that this approach cannot be continued forever. This led to the evolution of multi-core innovation. More and more applications now require higher performance out of the platform on which they are working. Processor architects are therefore moving towards multi-core processors. All kinds of integrated circuits and electronic systems including analog components, communications systems, logic devices, micro electromechanical systems (MEMS), sensors and microcontrollers are linked together to enable a safe and enjoyable driving experience. The use of single processing cores being replaced by multicore processors has opened up new challenges in the embedded domain. The code written for single processing cores and also the hardware peripherals being limited, has brought about the need for restructuring of the earlier code called the Legacy code.

Porting above mentioned legacy code to a new multi-core platform is now a preferred method instead of writing new code from scratch. One of the major challenges in migrating serial software to a parallel environment is to ensure that, the system's functionality is still correct after spreading the functionality across several cores, all executing simultaneously.

After the introduction of multi-core processors, software developers have started writing parallel programs for the effective use of available multiple cores. However, there are sequential legacy applications that have been developed over the past few decades. If such applications are being executed on multicore hardware, then optimal usage of all cores will not be guaranteed. Only one core will be utilized by such applications and the other cores would remain idle, if the operating system does not support any parallelism while scheduling. This is where it is necessary for the need of conversion of existing code which execute sequentially to one where the parallel architecture of multicores are utilized fully.

If an application's work can naturally be broken up, run in parallel, and aggregated at the end of an operation, such applications can be benefited from multi-core processing. Processor-intensive tasks such as video and audio processing, scientific and financial modeling applications and CAD rendering are examples of such applications. This work comprises of taking an existing application of automotive nature and effectively partition the application and its efficient porting on a multi-core platform.

2 Literature Survey

Advances in computing hardware provided significant increase in the execution speed of software with less effort from software developers. The arrival of multicore processors has provided the software developers with a new challenge that now they need to master the programming techniques to fully exploit the potential of multicore processing. First and fore-most important decision the developers must make when migrating to multicore is to select the appropriate form of multiprocessing for their application requirements which are basically, Asymmetric Multiprocessing (AMP), Symmetric Multiprocessing (SMP), and Bound Multiprocessing (BMP) [1]. This helps to determine the easiness with which maximum concurrency can be achieved for both new and existing codes.

The performance achieved by multi-core architectures was previously only provided by High Performance Computing (HPC) systems. The HPC programmers are re-

• *Jijinraj P is currently pursuing masters degree program in embedded systems engineering in amrita University, India,E-mail: jijinrajp@gmail.com*

quired to have a deep understanding of the hardware architecture in order to adjust the program explicitly for that hardware. This is not a suitable approach in embedded systems development, due to requirements on productivity, portability, maintainability and short time to market the product. The performance improvements of using multicore processors depend on the nature of the applications as well as the implementation of the software [2]. To take advantage of the concurrency offered by a multi-core architecture, appropriate algorithms have to be used to divide the software into tasks (threads) and schedule them to maximize the utilization of the processors.

Real-time systems can highly benefit from the multicore processors, as critical functionality can have dedicated cores and independent tasks can run concurrently to improve performance and thereby enable new functionality [3]. Moreover, since the cores are located on the same chip and typically have shared memory, communication between cores is very fast. Since embedded real-time systems are typically multithreaded, they are easier to adapt to multi-core than single-threaded, sequential programs, which needs to be parallelized into multiple threads to benefit from multi-core. If the tasks are independent, it is simply a matter of deciding on which core each task should execute (for embedded real-time systems, a static and manual assignment of cores is often preferred for predictability reasons.) However, many of today's existing "legacy" real-time systems are very large and complex, typically consisting of millions of lines of code which have been developed and maintained for many years. Due to the huge development investments, it is normally not an option to throw them away and to develop a new system from scratch. To benefit from multi-core processors, they therefore need to be migrated from single-core architectures to multicore architectures. The migration should maximize the performance without compromising correctness and quality attributes such as maintainability and portability [4].

Firstly, in order to compose high-performance applications, facilities from new hardware need to be exploited. Secondly, when additional cores are introduced in processor architecture, the speed of a single core may drop. Then, in order to maintain the existing performance, current applications need to be at least partially redesigned to benefit from parallel processing capabilities. These steps adequately reflect the activities –partial redesign, management of concurrency, and verification of the effect of parallelism that will be needed in practice, when porting existing sequential applications to multicore environment [5].

When porting legacy code from single-core systems to a multicore SMP system, or even when creating code for a multicore environment from scratch, the issues are:

- Code partioning from the single-core design to the multicore domain..
- Thread safety, Thread communication and synchronization.
- Probability of bugs like priority inversions, race conditions, deadlocks, etc.

Even if some of these problems can be prevented by an appropriate system design expertise, there is always the need to verify and track the results [6].

Frequent thread migration on a multicore system might happen if the number of threads queued for execution is much higher than the number of cores available for executing them. . In order to balance the load between available cores, the operating system might schedule threads to cores down to time-slice level. This potentially impacts performance, due to the overhead and cache misses.

To make the most efficient use of a multicore device, both hardware and software centric approaches are available. Hardware centric approaches are applicable to single-core systems and approaches like out-of-order execution (instruction reordering to avoid pipeline stalls) are used. Compared to the parallelism on instruction level, concepts such as single instruction multiple data (SIMD) and multiple instruction multiple data (MIMD) allows resource utilization on thread level [7]. From an application source code development perspective, these approaches are actually transparent. Software-related approaches comprise concepts like Symmetrical MultiProcessing (SMP), where a single operating system controls hardware resources then, dynamically assigns these to programs and threads which are to be executed simultaneously.

In parallel programming, both task and data parallelism are often considered to be mutually exclusive approaches. Still many applications benefit from both forms of parallelism. This is also the case in image processing. Image processing has found applications in many fields and applications, including the medical imaging, film and entertainment industry, weather forecasting, industrial manufacturing and inspection, etc. In some of these areas, the size of the images used are very large, yet the processing time has to be very short to keep up the timing constraints.. Sometimes real-time processing (keeping up with the frame rate of the camera) is required. As a result, there has been an increasing interest in the development and the use of parallel algorithms in image processing in the last decade [8]. Since image processing applications normally contain a lot of compute intensive data operations, data parallelism can offer better performance improvement than task parallelism

Data flow and data structure evaluations on the code provides scenario for the parallelization opportunities [9]. In particular, the data flow is helpful in determining whether the application exhibits functional or data parallelism. When there are a number of independent tasks that can run in parallel, the

application is well-suited for functional decomposition. When there are a large set of independent data that is handled through the same operation, the application is well-matched for data decomposition. OpenMP which is a compiler directed method, is intended to express data parallelism [10]. Thread libraries such as, POSIX threads (Pthreads) developed for Linux and other operating systems, are also useful. OpenMP and Pthreads are the most common packages used in the case of shared-memory system. For either functional or data parallelism, the programmer may write explicit threads to instruct the operating system to run these tasks concurrently.

When switching an application from single processor device to multicore, the first obvious solution is to write the different threads or functionalities from scratch. The designer can depend on, for example, the POSIX thread library for thread management to explicitly describe the data communication between threads [11]. In both scenarios, the designer is fully responsible for the parallelization. It is an error-prone and time-consuming task since the designer must add data communication and synchronization explicitly. This restrictions the exploration and usually means a far from optimal result even if it produces functionally correct solutions. A better approach is to create the parallel tasks using a high-level API (Application Programming Interface) such as OpenMP [12]. It naturally supports parallel execution of independent loop iterations or tasks. However, in many embedded application fields such as, wireless baseband processing and video coding, loop iterations and tasks are hardly completely independent and so need communication and synchronization between a subset of threads [13].

OpenMP provides some very useful APIs for parallelization, but it is the programmer's responsibility to identify a parallelization strategy and then use the relevant OpenMP APIs [14].Depending on the application code and the use-case, decision of what code snippets to be parallelized are taken. The 'omp parallel' construct, can essentially be used to parallelize any redundant function across cores. If the sequential code contains 'for' loops with a large number of iterations, the programmer can leverage the 'omp for' OpenMP construct that splits the 'for' loop iterations across cores.

3 Design and Implementation

3.1 Hardware

The Embedded Multicore Platform used is Freescale Semiconductor's IMX6Q Sabre Lite board. This board features Freescale's advanced implementation of the quad ARM Cortex™-A9 cores, which operates at speeds up to 1 GHz and having 1GB RAM. The multimedia performance of each processor is enhanced by a multilevel cache system, Neon MPE (Media Processor Engine) co-processor, a multi-standard (3D 1080p) hardware video codec, 2 autonomous and independent (2D and 3D) image processing units (IPU). Neon co-processor favours the SIMD (Single Instruction Multiple Data) processing, which will enhance the speed of processing.

3.2 Software

The board was already ported with Timesys Embedded Linux Operating System which was of the version Linux 3.0.15. By powering on the board, the Linux OS gets booted and makes the board ready for running the application. The execution of the application on the board is seen in the host PC through the serial terminal using the RS232 cable connected between the two. Minicom serial terminal is used for this purpose.

3.3 Application

Lane Departure Warning system (LDWS) is the application which is selected to run on the board. It is a Safety Critical Driver Assistance Automotive application. It is a real time image processing application which is used to alarm the drivers when the vehicle crosses the lanes. Basically the application consists of a series of modules which helps it to acquire the inputs, process it, manipulate and display the outputs. The application source code consisted of nine C files to incorporate the modules.

The application's input is the images taken while driving using the camera fitted on the vehicle. The images were taken in such a way that it contained too many lane crossovers in order to see if the application can give correct when executed on the multicore board. These images are the input to the LDWS application. After acquiring the inputs, the images are processed and the information in it are extracted. After validating the lanes, the corresponding warnings are generated.

3.4 Cross Compiler Toolchain

The cross compiler toolchains used to compile and build applications for this embedded board are Linaro Arm Toolchains which is gcc-arm-Linux-gnueabihf-4.7.3 and gcc-arm-Linux-gnueabi-4.6.2 . It is called as cross compiler because it compiles and builds the application in the host system (Desktop PC) and it is meant to run on the embedded target platform. The former one is a hard float ARM compiler which builds the application to a hardware related assembly code. This helps in using the inbuilt features in the embedded board such as Neon co-processor which helps in parallel execution of floating point operations. The latter is a soft float ARM compiler which uses the integer registers on the board for the floating point operations.

3.5 Implementation

When the application was considered for parallelizing, it was first decided to go for task parallelization ie., parallelizing the frames. Since the process is quad core, it can run 4 frames in parallel. But the problems came up because of the use of shared variables. A large memory was used for each frame and replicating it for 4 frames at a time was impossible

as far the embedded board is concerned. The second option was to parallelize the different processes done on each frame. It too caused a problem since the output of one module depends on another. So one has to wait for another for execution. Since it is an image processing application, there are large computations done with the data in the frames and it contributed for most of the execution time of the application. So loop parallelization was the best option and it was decided to be done.

For this, the functions containing the for loops, which are taking more time for time for execution is found by profiling using gprof. Then the for loops used inside those functions are analyzed to figure out whether they can be parallelized or not. The analysis include examining the data structures to assess whether parallelism may create data dependencies, causing incorrect execution or negatively impacting performance. One source of data dependency occurs when multiple threads write to the same data element. Without additional synchronization code, this can produce incorrect results.

By the detailed analysis of the application code and profiling, the for loops which needs to be parallelized to increase the performance is found out. It is parallelized using the OpenMP API since it favours data parallelism. It is done by putting the constructs above the for loops that can be parallelized. The variable that needs to be put in each clauses has been figured out using careful analysis. Otherwise, it will result in incorrect results. The parallelization was done for four for loops in the application source which was taking more time to execute.

Apart from the parallelization, the application was build with different compiler options and found the best combination which favoured the ultimate performance. The different options are:

1) Compiler toolchains
2) GCC Optimization flags
3) Neon co-processor in-built in the board.

The compiler toolchains used were gcc-arm-Linux-gnueabi-4.6.2 which is soft floating point ARM compiler and gcc-arm-Linux-gnueabihf-4.7.3 which is hard float. The application was executed with and without the optimization flag –O2. The other optimization flags were also checked, but –O1 & -O2 gave less performance and incorrect results respectively. So main focus was to check application performance on the board using the –O2 flag. Also the choice of Neon co-processor or Vector Floating Point (VFP) to use for the floating point calculations, were also analyzed by running the application with corresponding builds. VFP is the software floating point emulation.

4 Performance Evaluation

Both the sequential and parallelized application were executed on the desktop and board with various configurations and results are tabulated. The AlgoFPS denotes the speed of execution of algorithm in the application and AppFPS denotes speed of execution of the application.

Table 1
Performance of serial code on desktop

Compiler Version	CFLAGS (Type of compiler, opt levels)	GPU / Floating point processor	App FPS	Algo FPS
Gcc 4.4.3	NIL	NIL	17	18

FPS stands for Frames Per Second. CFLAGS are the compiler flags that are set in the makefile with which the application is built by the compiler. Tables 1 & 2 shows the performance of serial code on desktop and the performance of serial code and parallel code with different compiler options on the board respectively

From the experiments conducted, the configuration consisting of parallelized application, O2 optimization flag, hard float arm compiler, and Neon co-processor gave the best performance with a performance increase of approximately about 2.5 times considering the board and the PC. Comparing the baseline performance of board, the same configuration gave a performance increase of about 10 times. This experiment shows the performance improvement that was achieved by the effective use of code parallelization, choice of compilers, optimizations and the usage of inbuilt peripherals, for porting the automotive application into the embedded multicore platform.

5 Conclusion and Future Work

The legacy codes which are made to run on the single core machines can be effectively ported into embedded multicore platforms with large performance improvement by parallelization of the code, compilation parameters and the usage of inbuilt peripherals. Since the automotive applications comprising of image processing are being researched and developed, this work can be used as a reference for porting those applications into embedded multicore platforms in the vehicles for increased performance. The performance of the application on the embedded board can be further increased by offloading some of the code on to Graphics Processing Unit (GPU) available on the board which has numerous processing units inside. It will be challenging to figure out which and what portion of the code needs to be given to the CPU and GPU for better performance.

Table 2
Performance comparison with serial code & parallelized code on Board

Compiler Version	CFLAGS (Type of Compiler, Optimization Levels)	GPU / Floating Point Processor	Serial Code		Parallel Code		Result
			App FPS	Algo FPS	App FPS	Algo FPS	
Linaro Tool chain GCC version 4.7.3	O2, hard-fp	Neon	10	13	20	42	Matching
Linaro Tool chain GCC version 4.7.3	No optimization flags,Hard-fp	Neon	3	4	8	10	Matching
Linaro Tool chain GCC version 4.6.2	O2, soft-fp	VFPV3	10	13	9	12	Matching
Linaro Tool chain GCC version 4.6.2	No optimisation flags, soft-fp	VFPV3	3	4	8	9	Matching

References

[1]P.Leroux, R.Craig, "Migrating Legacy Applications to Multicore Processor", *Military Embedded Systems*.
http://www.mil-embedded.com/pdfs/QNX.Sum06.pdf, 2006.
[2] Nemati F, Kraft J, Nolte T, "Towards migrating legacy real-time systems to multi-core platforms" **,** IEEE International Conference on *Emerging Technologies and Factory Automation*, 2008, pp. 717 – 720.
[3] J.H. Anderson, "Real-time scheduling on multicore platforms", *In Real-Time and Embedded Technology and Applications Symposium*, 2006.
[4] W. Hwu, K. Keutzer, and T.G. Mattson, "The concurrency challenge", *IEEE Design and Test of Computers*, 2008, pp. 312-320.
[5] Seppanen A, Mikkonen T, "Porting Legacy Applications to Multicore: Experiences from an Industrial System", 17th Euromicro International Conference on *Parallel, Distributed and Network-based Processing*, 2009, pp. 127 – 132.
[6] Manfred Kreutzer,"Development of Complex Multicore Systems: Tracing Challenges and Concepts", *Mentor Graphics*, 2012.
[7] S. Singh ,"Challenges of programming multi-core microprocessors", *IET and Electronics Weekly Conference on Programmable Hardware Systems*,2008, pp. 1-29.
[8] Jie Zha,Yongmin Yang,Ge Li, "Real-time Image Processing System Based on Multi-core Processor", *Third International Symposium on Intelligent Information Technology Application*, 2009, vol.1, pp.329,332.
[9] Mignolet J.Y, Baert R, Ashby T.J, Avasare P, Hye-On Jang , Son J.C, "MPA: Parallelizing an Application onto a Multicore Platform Made Easy", *IEEE Micro* 2009, pp.31-39.
[10] Dagum L, Menon R, "OpenMP: An industry standard API for shared-memory programming", *IEEE Computational Science and Engineering* ,1998, pp. 46-55.
[11] Bob Kuhn, Paul Petersen, Eamonn O.T, "OpenMP versus Threading in C/C++", 2007.
[12] Diaz J, Munoz-Caro C, Nino A, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era ", *IEEE Transactions on Parallel and Distributed Systems*, 2012 , pp. 1369 – 1386.
[13] B. Chapman, G. Jost, and R. van der Pas, " Using OpenMP: Portable Shared Memory Parallel Programming", MIT Press, 2007.
[14] Mohan Rajagopalan, BrianT.Lewis,Todd.A.Anderson, "Thread Scheduling for multicore platforms"
http://www.usenix.org/event/hotos07/tech/fullpapers/rajagopalan.pdf,2007.