2015 The 5th International Workshop on Computer Science and Engineering
doi: 10.18178/wcse.2015.04.080

Asynchronous and Synchronous Approaches for Developing Software on Embedded Multicore Systems

Michael Bogner[1, a], Matthias Bonora[1, b] and Franz Wiesinger[1, c]

[1] University of Applied Sciences Upper Austria, Softwarepark 11, 4232 Hagenberg, Austria

[a]michael.bogner@fh-hagenberg.at, [b]matthias.bonora@fh-hagenberg.at,
[c]franz.wiesinger@fh-hagenberg.at

Keywords: multicore, MCAPI, MRAPI, C++11, threads, TBB, OpenMP, Boost, Poco

Abstract. This paper shows modern ways of developing software on embedded multicore systems. It sets its focus on established multithreading technologies on desktop systems and tests their use in an embedded environment. Existing multithreading libraries and newer developments are covered, with the main focus on solutions suitable for industry proven applications and requirements. Also, already established methods for using multicore embedded systems are examined, mainly the use of asynchronous multiprocessing. We compare these approaches, show their differences, examine their areas of use and test whether or not they meet the requirements in industrial applications. Our results show that the use of libraries, like C++11, Intel's TBB, Poco or Boost are able to meet the requirements while providing multithreading solutions for different use-cases, ranging from simple and basic parallelization up to low-level communication and optimization on chip level to larger, highly parallel applications. As the use of multicore processors will increase over time, we believe that these multithreaded libraries will be an essential factor in future development and that their use in embedded systems will be increasing accordingly.

Introduction

When observing the latest developments in software design in general and for embedded systems in particular a shift of paradigms is noticed. Requirements on performance, responsibility and power consumption are rising steadily and classical single-core processors are reaching their limitations. An increase of performance leads to higher frequencies and therefore higher power consumption, increased heat dissipation and realisation problems. To keep up with increased requirements, industry processors experience a major shift toward new platform technologies. An increasing number of hardware vendors introduces multicore architectures into their portfolio, promising a better trade-off between performance and power.

While in theory this shows a huge improvement, an adaption to these systems is not simple. Existing software has to be rewritten and verified in order to use the full potential of a multicore processor. It is tempting to avoid the multicore domain altogether. There are however major aspects which make the use of multicore architectures inevitable in near future. For once, energy efficient systems have to be equipped with multicore processors. The authors in [1] and [2] show that multicore architectures allow for more efficient execution, as each processor can be adjusted to performance needs. Single cores can be turned off or set into power saving mode when not needed but can be awakened when performance is required. This also concerns responsiveness of the system, another important aspect. Systems often require processing bursts for a short amount of time, while running at low power mode for long passages. Multicore architectures provide features to optimize such requirements. For example, ARM introduced a concept named big.LITTLE, where a high performance core is paired with a small lower-power core. Workload sharing across processing cores paired with the lower operating frequency and voltage of individual cores provides higher performance per watt and consumes less absolute power in low power mode as a single core CPU [7]. This concludes that the use of multicore systems will be essential for efficient systems.

As the embedded domain spans over a wide range of systems with different emphasis on constraints, not any embedded system type can be handled with the same approaches. This paper emphasizes mid- or higher range multicore processors, which run their software directly or use operating systems. Target areas may be embedded robotics, industrial control systems or automotive environments.

Related Work

The topic of multicore processors has been covered in a number of related publications, focusing mainly on power and performance aspects or specific software implementations.

The authors of [4] introduce the concept of asymmetric multiprocessors, where the workload can be handled by a powerful processing core while the rest of the system runs on a power efficient, smaller core.

Concerning specific implementations, the authors in [5] or [6] provide ideas on how multithreading can be implemented as streaming concept, using offline scheduling and specific tools for generating threads.

Our approach is to incorporate existing and tested software solutions and establish them in embedded environments.

Asynchronous Multiprocessing

A way of utilizing multiple cores on an embedded system is using asynchronous multiprocessing. Each core works as individual unit with as little inter-core communication as possible. Each unit runs independently and can be setup as needed, allowing the use of different operating systems or bare-metal builds simultaneously. The communication between cores is handled by a hypervisor.

Such a distribution has several advantages. It allows the use of different operating systems on a chip, like the combination of a real-time part for system control paired with an operating system for visualization. A huge advantage is that code parts can be reused with very little modification, allowing to adapt existing systems to multicore chips and adding additional features while using an already verified code base for critical system parts. Another way of using additional cores is as failsafe guaranty. The same code can be executed on several cores and results compared. Finally, asynchronous multiprocessing allows the utilization of heterogeneous multicore processors. As the code running on each core is predefined and not changing between cores, the different cores can work differently and even have completely different system architectures and instruction sets. This enables the combination of general purpose cores with highly specialized signal processing cores.

What is required by all those options is an interface for communication between cores. Here, recently introduced standards, defined by the Multicore Association, called MCAPI[8] and MRAPI[9] offer an interface and programming API designed to manage such inter-core communication. The standards define different ways of communicating between cores, which are contrary to each other. MCAPI centers on message passing, where data is exchanged explicitly via various different channels, similar to network sockets or streaming applications. MRAPI implements data exchange via shared memory, much like typical threading implementations, but extended to typical embedded requirements and features. The main distinction can be summarized as the way access to exchanged data is done. As different use-cases call for different synchronization methods, both approaches are useful in different scenarios. Of course, it is possible to mix both approaches within an application.

As versatile as asynchronous multiprocessing may be, its limits are reached when confronted with dynamic systems. It is not possible to distribute workload between cores, so tasks have to be scheduled offline at design-time. For systems consisting of many tasks with hardly predictable runtime such a segmentation may not be easily done. Also, single tasks cannot benefit of all processing cores, which reduces the peak-performance the whole system can offer. For such systems, synchronous multiprocessing approaches are better suited.

Dynamically Scheduled Systems

Dynamically scheduled systems are best suited for highly variant systems, where a multitude of parallel tasks is running for unspecified amounts of time. All cores are managed by a single operating system which manages lifetime and execution of each task, ideally giving all tasks their required processing time and keeping the system responsive to external events. This concept is also called synchronous multiprocessing (SMP). While task execution and scheduling is managed by the operating system, separation of a program into parallel segments is a responsibility of the software developer. To generate independent code segments, an interface between operating system and software has to be used. This can be done directly, or with the use of multithreading libraries. The efficiency of writing parallel code depends on how well a multithreading library is able to abstract required concepts. A well-developed library increases ease of use, functionality and encapsulation, allowing to write more efficient code. As most multithreading libraries are developed with desktop systems in mind, however, this has to be put into perspective. Although requirements for embedded and desktop areas overlap, there are restrictions with have to be considered and would render the use of a multithreading library impossible. These include the use of architecture specific features, footprint size of a library and tool chain support. Depending on the target system some libraries may not be applicable due to these restrictions, requiring the use of a library with a smaller feature set but better suited requirements.

Basically, there are two different approaches to introduce multithreading into a code segment: task parallelism, where the code is separated into independent sections which can be executed in parallel, and data parallelism, where each core runs the same code segment, but with a different set of data. A multithreading library should simplify the implementation of these two basic approaches as much as possible. This can be done by abstraction of these concepts and providing implementations for these approaches. Also, basic multithreading steps, like creating a thread, exchanging results or synchronizing code parts can be simplified by providing powerful templates and simplifications. The following section discusses different multithreading libraries and views how well they manage this integration and encapsulation.

Multithreading Libraries

This paper sets its demands to software which has to meet specific industry requirements. Applying these requirements to multithreading libraries, the following selection could be made: Boost thread libraries [11], Poco thread libraries [12], OpenMP [13] and Intel's Thread Building Blocks [14]. All these libraries are established in desktop environments and used in several software projects. In addition, the newly introduced C++11 threading addition to the standard library[10] will be viewed and tested how well it can compete with or supplement existing multithreading implementations.
With the release of C++11 extensions for multi-threaded software development are introduced. This includes a multithreading aware memory model as well as extensions to the standard library. Being standardized, it is guaranteed that the used functions are working properly with different system architectures and compilers. As applications may target several different operating systems and architectures, this is a valuable addition.

Viewing the feature set of C++11, it becomes apparent that the release of a feature-rich library with advanced features was not intended. Though the memory model is well designed and has several options for low level optimization, the level of abstraction is kept low. In addition to basic thread and synchronization functions, a simple abstraction for asynchronous execution is provided. These so-called futures let the developer describe a function including return value which runs asynchronously from the main processing thread. Synchronization is done when the result of the asynchronous execution is required; the caller blocks until the external function finishes execution and returns the value.

A downside of futures is the extensive creation of threads when spawning an asynchronous task. Here, additional functionality and advanced features would be required. This also projects to the

whole library. While all basic concepts are available, a richer feature set is required. However, this affects the footprint of the library, as when using C++11 threads, code size is the smallest of all libraries.

For the Poco or Boost libraries the functionality is similar to that of C++11. They had provided a threading interface for C++ long before C++11 was released. In fact, the C++11 interface is heavily influenced by the Boost threading library. What is added by Poco and Boost is support for a wider range of compiler and operating systems, as they don't dependent on a modern compiler supporting the new standard. In the case of Poco libraries, a few higher level features are added, like task management or thread pools. However, real abstraction for simple multithreading is still not provided. The downside of using Poco or Boost is a slightly larger footprint compared to C++11.

Adding real advanced feature sets for multithreading software, two major libraries can be noted: The OpenMP API specification and Intel's Thread Building Blocks (TBB). Both aim at simplifying the creation and management of multiple threads. Their goal is to let the developers describe which part of the code they want to run in parallel in a simple way, without the need of adding information on how threads are created, synchronization is done or results are collected. While having the same goal, the approaches of TBB and OpenMP are quite different. TBB depends heavily on C++ templates and language features while OpenMP uses pragma defines and Compiler extensions to implement their specification. Both approaches have their advantages and disadvantages. A C++ only implementation is easier to port to different platforms, as the library itself is independent from compilers. On the other hand, a compiler based approach offers a wider range of optimization, up to the point where code could be distributed not only on CPU cores, but also on graphic processors.
This reflects on the current state of implementation on embedded systems. Taking the Android platform for example, Intel has announced experimental support for the Native Toolkit (NDK). For OpenMP, support depends heavily on compiler and tool chain vendors. It is likely that existing tool chains have to be upgraded in order to support OpenMP. It is planned however to add accelerators and architecture features, like SIMD vector technologies to the next draft of OpenMP. This would increase the value of OpenMP significantly when also applied to embedded systems.

To conclude, both TBB and OpenMP have major potential. They offer an advanced set of features which is far superior to those of the other libraries. However, as platform support is still not very well established yet the use of these libraries has to be considered carefully.

Evaluation

Which library is the best for a specific system depends on the systems' specification. Lightweight systems will most likely benefit from the small footprint of C++11, as flash size may be of importance. When older compilers are used, Boost or Poco libraries are a good replacement. Here, Poco may be better suited, as it supports several hardware platforms and operating systems officially.

An advanced set of features is only provided by TBB and OpenMP. Especially data parallel execution is introduced easily while the library handles efficient data segmentation. The downside is a larger footprint. Also, when working with real-time requirements the automatic segmentation and workload distribution is an offset, as it makes timing and runtime behavior tasks not predictable. The biggest drawback is that both TBB and OpenMP have only limited availability on embedded systems, requiring unofficial ports, beta versions or new compilers and tool chains in order to work, making them only a potential option for current use.

Conclusion

Our evaluation shows that current technologies for writing multithreaded code on embedded systems are far from well established. Support for programming libraries is only starting to emerge. While research projects and vendor announcements promise improvements, it will still take time until embedded systems can be designed for embedded multicores as easily as desktop or server systems.

Multicore processors in embedded systems, however, is a trend which will definitely be increasing over time, as there is no way around them.

A good workaround is the use of asynchronous multiprocessing, as it allows the reuse of existing, tested software and allows the developer to add new features independently. This does not work in all cases however, and reduces the potential a multicore system could offer. A possible alternative is the combination of both worlds. When the number of available cores is sufficient and the used hypervisor offers the option, the whole system can be segmented in multiple SMP subsystems, where each subsystem runs on one or more cores. When a good separation is found, this may offer the best compromise between performance and usability. Parts which run on a single core can be used without alteration. What is still required is the adaption of those software parts which run on the SMP subsystem.

So, adapting to multicore programming is a task not optional but mandatory. Having understood the need, adaptations and libraries are pushed forward with increasing speed, trying to close the gap between embedded and desktop implementations. This will pave the way for software developers to write efficient, portable and maintainable code for multicore systems.

References

[1] C. H. (Kees) van Berkel. 2009. Multi-core for mobile phones, (DATE '09). European Design and Automation Association, 3001 Leuven, Belgium, 1260-1265.

[2] Marcu, M., Tudor D., Fuicu S. et al.: Power efficiency study of multi-threading applications for multi-core mobile systems. W. Trans. on Comp. 7, 12 (December 2008), 1875-1885.

[3] Becchi, M. C. (2006). Dynamic thread assignment on heterogeneous multiprocessor architectures. New York: ACM, 2008

[4] Choi, Y., Lin, Y.,Chong, N.,Mahlke, S., Mudge, T.: Stream Compilation for Real-Time Embedded Multicore Systems, CGO '09, p.210-220, 2009

[5] Gordon, M. I., Thies W., Amarasinghe S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs, ASPLOS-XII. ACM (2006)

[6] NVIDIA Corporation: The Benefits of Quad Core CPUs in Mobile Devices, 2011, www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf

[7] The Multicore Association: Multicore Communications API (MCAPI) Specification, 2011

[8] The Multicore Association: Multicore Resource API (MRAPI) Specification, 2010

[9] ISO, 2011. ISO/IEC 14882:2011: Information technology - Programming languages - C++. Available from: www.iso.org

[10] Boost C++ libraries, www.boost.org

[11] POCO C++ Libraries, pocoproject.org

[12] OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 3.1, July 2011, www.openmp.org/mp_documents/OpenMP3.1.pdf

[13] Intel Corporation: Intel(R) Threading Building Blocks Reference Manual. 2011