2014 13th International Symposium on Parallel and Distributed Computing

From MultiTask to MultiCore: Design and Implementation Using an RTOS

Célio Estevan Morón, Antonio Ideguchi, Marcio Merino Fernandes
Departamento de Computação
Universidade Federal de São Carlos
São Carlos - SP, Brazil
Email: celio@dc.ufscar.br

Allen D. Malony
Depart. of Computer & Inf. Sci.
University of Oregon
Eugene, OR, USA
Email: malony@cs.uoregon.edu

***Abstract*—Practice has shown that programming a new multicore system is a greater challenge than previously thought. The challenge is to produce the resulting system in a way, which is as easy as sequential programming. This new trend has changed the way we think about the whole development process. The aim of this work is to show that it is possible to develop a multicore embedded system application using existing tools, while at the same time, obtaining reuse. This process is carried out in a cyclic and increasing manner, generating a more refined version of the application at each iteration. The development process consists of five phases: Multitask Modelling, Code Generation, Test/Debugging, Mapping Tasks to Cores and Tuning the Application. The three initial ones are carried out using the VisualRTXC tool, whereas the last two use the performance tool TAU. Using a small application, a Case Study shows how the proposed development process works and the steps involved in the implementation of an embedded system.**

I. Introduction

Be it in the form of specialized high-performance systems, or dedicated embedded systems, multiprocessor machines have been present among us for decades. The concepts involved in the programming process of multicore systems are quite well known in the fields of Operating Systems and Parallel Programming.

However, so far there seems to be no tool available which could make the programming process of these systems as easy as it is for sequential programming. In order to be able to program multicore systems efficiently, we need joined knowledge from: (1) Parallel Programming; Mechanisms of Communication and Synchronization of Process; (2) the Processor's architectural details; and (3) Performance evaluation tools. To a certain extent, knowledge in these subjects is already used in parallel programming and real-time systems. In the real-time field to fulfill the time constraints, while in parallel programming to try load balance and to avoid bottlenecks.

Basically, there are two techniques to program multicore systems: by using Multi-processes or by using Multi-threads. Multi-processes enables to run multiple processes at the same time. In single-processor architectures one process is assigned to the CPU, whereas in multicore architectures one process can be assigned to each core in order to extract the performance benefits from parallelism. For embedded applications, the process is called task, and multitasking is a key technique that can lead to substantial performance improvements and/or cost reductions.

Multithreading is a technique that allows users to obtain performance benefits of general multicore processors. However, multithreading requires applications to be designed in such a way that the work can be completed by independent workers, acting in the same process. The different threads can be allocated to run in different cores. Threads are sequential processes that share memory. When the execution modules are independent, threads can be used efficiently. However, it is not the way typical embedded systems work. Threads make programs highly nondeterministic and rely on programming styles to constrain nondeterminism in order to achieve deterministic aims [1].

Embedded systems are developed mainly using the multitasking technique. The term embedded system [2] refers to any computer system built within a device and working as part of it. Most embedded systems have real-time features associated to them. Embedded systems using microcontrollers typically rely on a Real-Time Operating System (RTOS) to provide multitasking capabilities. RTOSs improve performance and enable more sophisticated programs on less expensive processors.

Developing a multitask system is inherently complex, since it involves synchronization among tasks and data dependence analysis. Multitask systems consist of several processes, called tasks, which depend on each other to execute properly. In order to create these systems, the developer needs to split the application modules into a group of tasks, which are able to run simultaneously. In addition, the application of efficient methods for communication and synchronization is essential in order to ensure that the processes interact correctly.

However, most tools used for developing multitask applications are poorly adapted to support the additional complexity of these systems. As a result, developing multitask applications is frequently associated to various challenges, because the tools used for their construction do not usually follow adequate software engineering techniques. In many cases, the only tools available for the developer are basic software-like compilers and text editors. Developing a system using multicore processors follows the same multitask approach. Several stages of the development life cycle are more complex in multicore systems than in traditional applications, and

978-0-7695-5265-1/14 $31.00 © 2014 IEEE
DOI 10.1109/ISPDC.2014.18

therefore the use of adequate tools to provide support for multicore programming is a real need.

Recent works [3], [4], [5], [6], [7], [8] show that under some conditions message passing could cost less than shared memory in the multicore domain. Lauer and Needham [9] showed that message passing and shared-memory are duals, and the best choice depends on the architecture. It seems that so far the PC architecture has favored the shared memory model but it cannot provide for scalability.

The traditional development of embedded systems based on RTOSs splits the application into tasks that are executed concurrently. Consequently, by using an RTOS, we take the partitioning of the application for granted when we go multicore. The approach being proposed here is to mix the traditional RTOS with a communication channel [10] in order to achieve simplicity and reuse in the development of multicore systems. By following our model, if an application can be programmed in the form of multitasks then it is possible to run it also in a multicore architecture. RTOSs exchange information among tasks using several kinds of queues. In order to generate the multicore program we simply exchange the multitask queue for a multicore channel. In our implementation the parameters are the same in both cases, needing only to change the primitive's name. Figure 1 shows our approach which is based in the RTOS multitask structure plus communication channels.

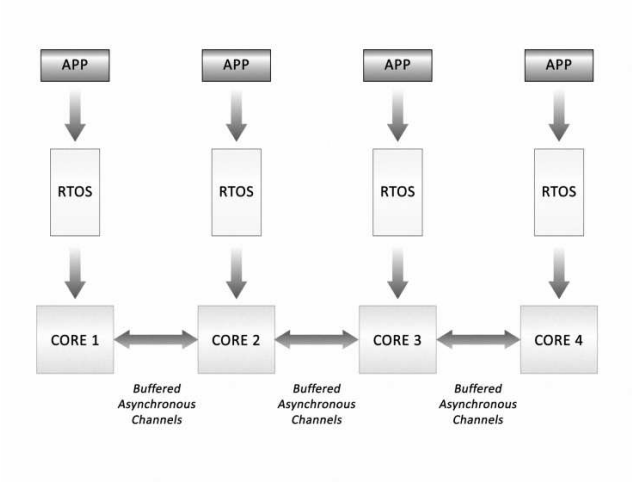

Fig. 1. RTOS MultiTask plus Communication Channels.

A relevant contribution of this work is to facilitate the development of multicore embedded systems using a visual environment. To fine-tune a parallel application, there is a need for a tool that reflects its parallel structure. In a multicore architecture, the programmer has to be aware of the underlying hardware in order to map it into physical entities. A performance tool is used to help in the work of fine-tuning the mapping of tasks into cores.

This paper presents a development method for multicore systems using a graphical productivity tool for embedded real-time applications based on multitasking. The paper is organized as follows: Section 2 presents related work: a Visual Environment, Debugging and Testing Tools, Performance Analysis Tools, the Traditional Development Cycle. Section 3 provides an overview of the tools used in our development process: VisualRTXC Implementation Tool and TAU Performance Tool. Section 4 presents a method for developing multicore embedded systems. Section 5 presents a Case Study (Mandelbrot Set). Finally, the conclusion is discussed in Section 6.

II. Related Work

In this section, we will discuss some of the subjects included in the work presented in this paper.

A. Visual Environments

A considerable amount of research in the area of visual environments has been carried out aimed at developing software able to reduce the difficulties found in the development of parallel systems. This effort resulted in the development of user-friendly tools such as TEV [11], PVMGraph [12], Millipede [13], TRAPPER [14] and P-GRADE [15]. Although there are several differences between these environments, they all focus on making the program development easier, providing graphical representations to map paradigms or libraries designed for developing parallel systems.

TEV (Teaching Environment for Virtuoso) is a Visual Environment for the Development of Parallel Real-Time programs, a tool whose aim is to facilitate the generation of source code of applications developed for the Parallel Kernel Virtuoso. This Visual Environment provides continuity when developing projects using the most common development methods (traditional or object-oriented), by offering support during the implementation, debugging and testing phases.

Despite the benefits provided by the tools mentioned above, it is clear that they were not designed to support the development of embedded multicore systems. These environments were rather designed for multitask systems based on workstations or workstation clusters. In these architectures, the processing nodes are distinct machines, interconnected through a high-speed network and running general-purpose operating systems.

B. Debugging and Testing Tools

Due to concurrency, multitask and high performance issues, embedded multicore systems have always presented an additional difficulty during debugging. This problem is not new; nowadays not only specialized people from the embedded or high performance field need to go through this, but also every programmer building a system. This poses huge new challenges in the debugging and testing process of multicore programs. The non-determinism issue makes it very difficult to reproduce an error. An efficient approach to diminish errors is to identify the concurrency as early as possible.

In addition to early bug detection and removal techniques, the final phases of testing (function, system, and stress) are enhanced by making the tests a lot more likely to exhibit

existing bugs in the code. This procedure is carried out by changing the internal timing of the executions in a way that is more likely to show abnormal behaviors.

C. Performance Analysis Tools

Performance analysis and evaluation is a very challenging process in multicore systems. Porting a single thread program to run in a multicore system is even more difficult. Program instrumentation and tracing are the most commonly used techniques to obtain program execution profile for analyzing the behavior of the program and its performance bottlenecks.

The performance of the application is heavily influenced by the thread interaction. As the level of parallelism increases, resource locks occur more frequently, possibly requiring additional application redesign. Usually, the correct design of communication and synchronization protocols is a critical phase in order to achieve high multi-threaded performance. Finally, performance is influenced even when there is no communication, i.e., processes running in parallel can affect each other's performance, for example by accessing the cache.

A number of performance tools were developed for High Performance Computing (HPC) and can be used with multicore systems. Some of those tools include TAU [16], HPC-Toolkit [17], and Paraver [18]. In the non-HPC arena, Intel's VTune [19] is probably the best performance tool for multi-threaded programs, where its critical path analysis is a particularly useful feature.

Trace-based analysis and visualization can help performance investigation of multi-threaded programs running on multicore systems [20]. The problem with the traditional approaches, such as profiling, is that data aggregation makes it impossible to understand precisely those communication issues that currently become so important. However, a serious issue with timeline data is its lack of scalability, making it difficult to manipulate and visualize it, and in this way for the user to easily grasp where to focus the investigation of performance bottlenecks. The growing number of cores and threads created by the applications makes it even more difficult to visualize all the threads and/or cores at the same time.

D. Traditional Development Cycle

Traditionally, the development process of embedded/real-time applications has been aided by a set of tools. This development process is iterative and is completed when the application requirements are met. The four steps involved in this process can be summarized as:

- **Analysis**: for determining how to divide the application into tasks or threads;
- **Design**: for deciding how to implement the tasks/threads;
- **Implementation**: for implementing the source code;
- **Debug**: for finding and fixing task/thread-related bugs in the code (Data race, Thread stall and Deadlock);

In this proposal, the development cycle for the design of multicore systems can be carried out using a graphical tool to define all parallel activities in the application (i.e., all process management, communication, synchronization, and so on). Graphical support can make the debugging and testing of parallel applications easier if the programmer can create the graphical representation of the application during the development phase. There are several advantages in using this approach: (1) people who are familiar only with sequential programming can easily use the proposed method to develop multicore applications in order to exploit the available computational power in an efficient manner; (2) the debugging phase of the software development process can be simplified by the use of the graphical tool instead of ordinary textual source level; (3) debugging large parallel programs is much more difficult than sequential ones, but using a graphical tool allows the user to divide it into different layers; (4) when the user needs to detect an error related to the parallel part of the program, this can be better visualized using a graphical environment.

For multicore systems, it is necessary to have an additional step in the cycle of development; that is, the tune of the application. For this, a performance tool is necessary in order to carry out the optimization of task performance.

In particular, the last step is very important for the overall performance of the resulting system. It is important to know how much of the application is running in parallel; is the work evenly distributed between tasks? What is the impact of synchronization between tasks on execution time? Is memory being used effectively and shared between tasks and processor cores? And so on.

III. TOOLS USED IN THE DEVELOPMENT PROCESS

Our development method is based on two tools: VisualRTXC and TAU. A short explanation of them follows below.

A. VisualRTXC Implementation Tool

VisualRTXC is a tool that generates code to be executed by the real-time kernel RTXC from Quadros Systems Inc. (www.quadros.com). Both, the visual environment and the RTXC simulator can be downloaded from http://www.quadrosbrasil.com.br/html/en/RTXCsim.php.

Developing embedded systems is frequently associated to various challenges, mainly because the tools used for their construction do not usually use modern software engineering techniques. Because many problems, mainly in the area of real-time systems, are inherently concurrent, using better tools to provide support for multitask programming is a real need.

A large amount of research in the area of visual environments has been carried out aimed at developing software able to reduce the difficulties found in developing parallel systems. By using these tools, it is possible to specify the parallelism of the program at a high level of abstraction and from which the source code can be automatically generated. In the case of embedded systems, the aim of using development tools was to maximize the performance of the hardware rather than to improve the user's productivity. As a consequence of this approach, there are many difficulties concerning the development of this kind of system due to the limitations imposed by these tools. In most of these cases, the user is

provided just with a text editor and a compiler. However, it is evident that an embedded system that is organized simply as a set of text files cannot be easily understood nor managed. The final stages of the life cycle of these systems are usually quite complex and lack adequate tools.

VisualRTXC is a graphical tool designed to help develop, document and visualize embedded applications. It can be thought of as a layer above the services offered by real-time kernels, acting mainly over basic structures used by these kernels such as tasks, semaphores, resources, timers and others.

Unlike the traditional approach, where the program implementation is carried out on the source code, VisualRTXC offers a higher abstraction layer, where it is possible to graphically represent most of the embedded application characteristics.

The graphical representation is based on the use of graphs, a widely used concept for the development of programs to describe process structures, data dependency, performance visualization, and so on. In VisualRTXC, graphs are used to describe the application sketch, while the processes' low-level details are defined through a combination of textual and graphical notations.

By providing separate notations to describe the application structure and the tasks' algorithm details, VisualRTXC allows the software engineer to divide the embedded program into three abstraction levels, each of them suitable for a specific stage of the implementation process.

- **Application level**: it is the highest level of the embedded application and helps during the initial development phase, offering a general view of the program structure. The basic structures provided by the kernel (tasks and other microkernel objects) are described graphically, while the functionality of the tasks (source code) is omitted. Tasks are seen as "black boxes" and only the exchange of messages among them is shown.
- **Task level**: graphically describes the source code of the tasks, emphasizing the flow control and the exchange of messages among them. A separate diagram is produced for each task created at the application level. Loops, conditional structures and operations for sending and receiving messages are represented through icons defined by the tool. A special symbol is defined to graphically denote the textual blocks, which are portions of code without graphical representation.
- **Textual code level**: the lowest level of the application, where the programmer can define fragments of code associated to the symbols that represent the textual blocks at the task level. In these fragments, portions of source code, not directly related to the communication and synchronization of tasks (variables, functions, attributions) are inserted in the traditional manner, using a text editor provided by the tool.

As VisualRTXC allows the user to divide the application into several layers, it is possible to run each layer code on a specific core.

B. TAU Performance Tool

The TAU parallel performance system is the product of fourteen years of development to create a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications. The success of the TAU project represents the combined efforts of researchers at the University of Oregon and colleagues at the Research Centre Juelich and Los Alamos National Laboratory. The TAU (Tuning and Analysis Utilities) can be downloaded from http://www.cs.uoregon.edu/research/tau.

TAU provides an API that allows programmers to manually annotate the source code of the program. Source level instrumentation can be placed at any point in the program and it allows a direct association between language and program-level semantics and performance measurements. Using cross-language bindings, TAU provides its API in C++, C, Fortran, Java, and Python languages.

The performance tool TAU is very helpful at least in three situations: 1- Optimization of the generated source code; 2- Analysis of bottlenecks during execution; 3-Trade-off between performance and source code modifications.

The integration of TAU in the development process has made it possible to adjust the resulting source code to the best possible performance for the working architecture.

IV. DEVELOPING MULTICORE EMBEDDED SYSTEMS

Our development method starts implementing the applications in the form of a multitask system and guaranteeing that after the tuning phase it will also execute correctly in a multicore architecture.

We are proposing a development process that can be carried out in a cyclic way generating more refined versions of the application at each iteration. Within each cycle, the tasks are categorized into five phases: Multitask Modeling, Code Generation, Test/Debugging, Mapping Tasks to Cores and Tuning the Application. The software engineer may use any analysis and design method in conjunction with the proposed development process, including object-oriented methods such as UML, or even other approaches.

VisualRTXC [21] and TAU [16] can help systems programming by providing an intuitive user interface and high-level design objects that are tightly coupled to the underlying kernel architecture. This development process allows the developer to rapidly move between design concepts and generated C code. In addition, it provides visual abstraction and design help for each of the typical phases of the development life cycle.

Representing applications using graphs is another advantage offered by a tool such as VisualRTXC, since this approach is familiar to most designers and makes it easier to use the services provided by commercial kernels. As a result, the productivity of the development team is highly increased and as a consequence, better results can be obtained in less time at lower costs.

The possibility of quick experimentation makes the proposed development process quite adequate to create prototypes of the multicore embedded application during the initial stages of development. By creating prototypes, problems existing in the design can be detected and fixed early, minimizing the consequences of them.

In addition, the division of the graphical representation into layers allows the application to be structured as a hierarchy of subsystems. This modular approach makes our method ideal for developing large multicore embedded applications, where it is impossible to represent the whole system within a single diagram.

The combination between the proposed model-driven process development and the tools VisualRTXC and TAU also makes it easier to integrate different tools into a single programming environment, allowing the same graphical notation to be used by the environment tools throughout the different development stages.

A. Phase 1 - Multitask Modeling

Beginning with the application requirements, the software engineer models the application using VisualRTXC and the techniques available on the embedded domain. The modeling step is carried out at two levels. First, at the system level, where the software engineer represents the executing modules (tasks, RTXC threads and exceptions) and their kernel primitives, as well as all communications and synchronizations. The second level refers to modeling the executing modules, and is carried out after the first level of modeling has been made explicit.

The two levels of modeling are carried out using VisualRTXC, a tool that has a user-friendly graphical interface, having dockable windows and displaying multiple documents at the same time, with each document being shown within its own window.

Using the Integrated Development Environment provided by VisualRTXC, the software engineer develops a graphical model to represent the embedded application. This graphical model can be complemented with textual descriptions (segments of source code created by the software engineer). From this information, VisualRTXC automatically generates the source code of the application, which can be easily built and run.

B. Phase 2 - Code Generation

As modeling is carried out at the system level and at the executing modules, the source code associated is generated automatically. The model drives the process to generate the code of the application.

This application model-driven architecture is refined to complement the generated code with the C commands that are not related with the kernel primitives.

The next step complements the code that was automatically generated. During this phase, VisualRTXC helps to add new elements on the modeling at the executing modules level.

VisualRTXC provides the following features to facilitate the implementation step: view synchronization, export/import interface, multi-developer environment and reverse engineering support.

C. Phase 3 - Test and Debugging

A significant limitation for developing embedded systems is the lack of adequate programming tools, mainly those needed to support the final steps of the life cycle. The proposed development process, whose main aim is to facilitate the implementation, debugging and testing of multicore embedded applications, integrates the three basic components (behavioral, structural and functional), into a single graphical representation, facilitating, therefore, the understanding of the system as a whole.

The testing and debugging processes of the system during development is made easier by the comprehensive view provided by the modeling at the system and at the executing modules levels. Our experience has shown that debugging time is considerably reduced by applying the development process proposed here.

A major problem found in developing embedded systems is the inherent difficulty to produce rapid prototypes of the application. In addition, it is not uncommon for the development of these kinds of systems to be behind schedule. By linking Design, Implementation and Debugging in the proposed development process, the user is able to produce a rapid prototype that considerably shortens the development process.

Following this development process, the software engineer can run the produced prototype on a single-processor architecture and return to the initial step until the behavior meets the requirements, making the necessary adjustments throughout the steps. This process is repeated until all the requirements are fulfilled. At the end of this process we have obtained from the TAU tool a set of performance diagrams (obtained using a single processor) which provide an initial estimate of the computational power demanded for each task and function.

D. Phase 4 - Mapping Tasks into Cores

At this stage, the application has already been split into tasks and run in one core on the x86 architecture host. There is also a performance diagram including information that allows mapping tasks into cores. Usually, this mapping is carried out with load balancing in mind, but could also be used to ensure the fulfillment of the real-time constraints for specific tasks.

At this point, one executable is generated for each core containing the set of assigned tasks. Moreover, the RTOS queue is exchanged for some kind of communication channel (usually pipes) at every inter-core communication.

The next step is concerned with loading all the executables to run on the cores and proceeding with the tuning of the application.

E. Phase 5 - Tuning the Application

To truly understand the interaction among multiple tasks running on different cores simultaneously, one must be able

to examine how the application interacts with the whole system. The primary benefit of using a performance tool is two-fold: understanding how an application is actually performing; and, identifying bottlenecks in the development process. Performance analysis can be used to ensure that real-time specifications are met.

Instrumentation identifies every execution of a particular function. Therefore, this phase is aimed at verifying whether the mapping, carried out based on a single core execution, was successful or not when running in a multicore architecture. If necessary, returning to Phase 4 and repeating the mapping process in order to achieve the system's goal.

V. CASE STUDY: MANDELBROT SET

Code that generates the Mandelbrot set is a favorite target for evaluating performance in embedded systems. Embedded systems usually require a high amount of image processing to perform and the Mandelbrot set can be adjusted to demand the computer power necessary for evaluation. Additionally, the Mandelbrot set is familiar to most users and the code required to generate the images is quite simple. The Mandelbrot Set is a set of complex numbers that show interesting behavior when run through a simple formula. When this formula is applied iteratively (with the result from one calculation used as the input for the next one), numbers within the Mandelbrot set will not reach further than a certain distance, regardless of how many times the formula is applied. Numbers not in the Set will exceed that magic distance, after some number of iterations.

A. Multitask Modeling

Starting with the application requirements, the software engineer models the application using VisualRTXC. The modeling step is carried out at two levels: the system level and the executing modules.

The development environment is organized into four distinct areas: Workspace Window, Layer Diagram, Code Diagrams and Output Window. Figure 2 shows the Layer Diagram and the Code Diagram for the Mandelbrot Set application. It can be easily seen in the Layer Diagram that the MASTER task will generate blocks of XBLOCK by YBLOCK dots and place it in the COMPQ queue. WORKER1 and WORKER2 that will run on the same core (multitask version) will take the blocks in the COMPQ queue to work on. They will call CAL_PIXEL to calculate the dots to be plotted and place them on the COLQ queue. Finally, the task PLOTTER will obtain the calculated dots from COLQ queue and plot them.

B. Code Generation

The code was instrumented with commands from the performance tool TAU. The output of TAU can be seen in Figure 3. It shows the mean time for each executing module. The call are inclusive which means that main( ) will call MASTER, PLOTTER, WORKER1 and WORKER2. In the same way, WORKER1 and WORKER2 will call CAL_PIXEL. As the points to be calculated is divided between WORKER1

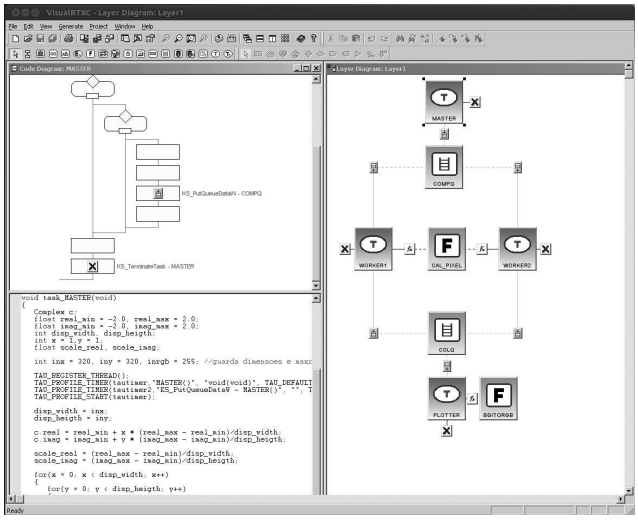

Fig. 2. Code Diagram and the Layer Diagram.

and WORKER2, from Figure 3 it is possible to conclude that the execution of WORKER2 is having precedence upon WORKER1 (CAL_PIXEL time of WORKER2 is approximately half of the time of WORKER1). It is also possible to see that MASTER is being delayed because queue COMPQ is full (MASTER-WriteQueue). The same happens to PLOTTER which is able to get the point from queue COLQ at a faster rate than WORKER1 and WORKER2 can process it.

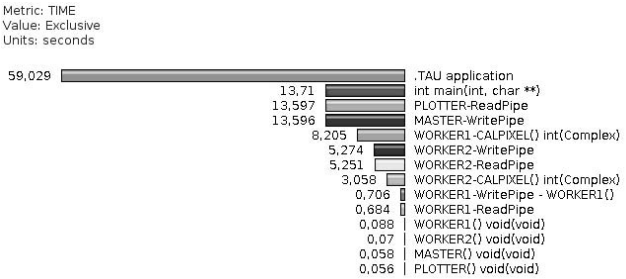

Fig. 3. TAU ParaProf Multitask.

The generated code was executed using the Real-Time Kernel RTXC running on top of Ubuntu 11.04 in a Processador Intel(R) Core(TM) i7-2600 CPU @3.40GHz 3.40 GHz, 16GB RAM Memory. The result was a file with the Mandelbrot Set and profiles that generated the graph shown in Figure 3.

C. Mapping Tasks to Cores

So far Phase 1, Phase 2 and Phase 3 have been carried out. The application was modeled in concurrent tasks, and the corresponding code generated, tested and debugged. Furthermore, the execution time of the tasks is in a graphical representation. At this point we move from the host based on x86 architecture to the target architecture. We tested the method with two targets: 8 core on x86 architecture and 2

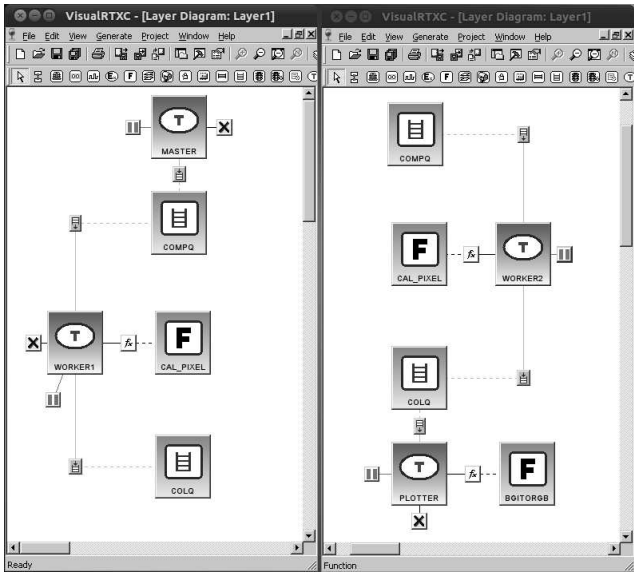

Fig. 4. - Layer Diagrams for Core 1 and for Core 2.

cores on a Arm architecture. Based on the performance graph, we chose to place the tasks MASTER and WORKER1 in core 1; tasks WORKER2 and PLOTTER in core2 on the Arm architecture. From the multitask version of the Mandelbrot Set, one executable was generated for each core. Figure 4 shows the two sub-projects generated for the dual core PandaBoard target. Figure 5 shows how easy is to move from multitask to multicore. All inter-core communication is replaced by a communication based on the Linux pipe.

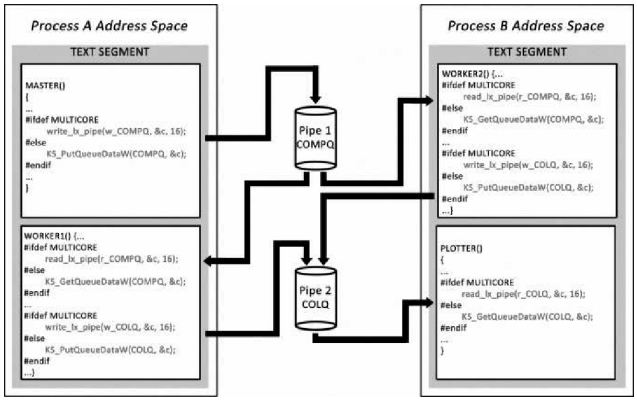

Fig. 5. RTXC queue versus Linux pipes.

D. Tuning

This phase verifies whether the mapping of tasks based on the multitask execution times provides the aimed results. Figure 6 shows the performance graph of the tasks running on the 8 cores implementation. It can be seen that the execution time on the 8 cores is fairly balanced.

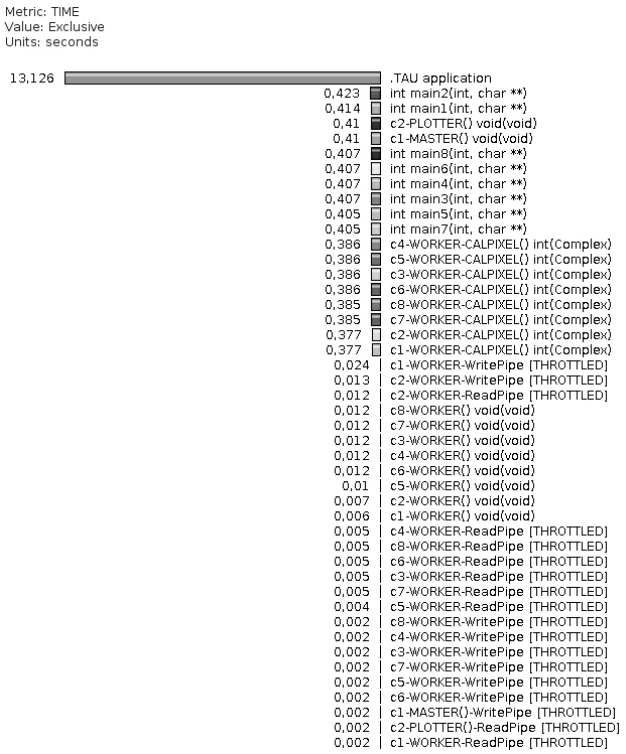

Fig. 6. TAU ParaProf MultiCore 8 Cores-x86.

Both executions, on Arm and x86 archictetures, were processed for a Mandelbrot image of 1280 x 1280 points.

TABLE I
TOTAL EXECUTION TIME ON ARM ARCHITECTURE

Mandelbrot Set	RTXC Multitask	RTXC Dual Core
ARM	1176.56s	442.48s

The Arm architecture execution was obtained using a PandaBoard based on the processor OMAP4430, a Dual-core ARM® Cortex-A9 MPCore with Symmetric Multiprocessing (SMP) at 1 GHz each.

TABLE II
TOTAL EXECUTION TIME ON X86 ARCHITECTURE

Mandelbrot Set	RTXC			
	Multitask	2 Cores	4 Cores	8 Cores
X86	118.76s	53.14s	26.52s	20.37s

The 8 cores x86 architecture execution was obtained on a Intel(R) Core(TM) i7-2600 CPU @3.40GHz , 16GB RAM Memory.

Table I shows the total execution time for the 2 cores ARM architecture. Table 2 shows the total execution time for up to 8 cores on the x86 architecture. From Table II was generated the graph on Figure 7 showing the speed-up for the x86 architecture.

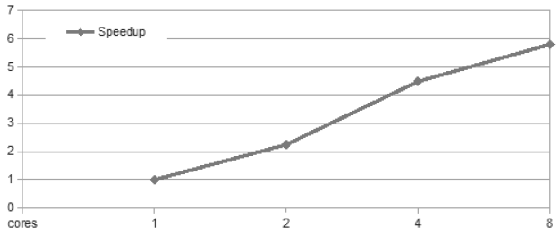

Fig. 7. Speedup for 8 cores.

VI. CONCLUSIONS

The exclusive use of general purpose tools, such as compilers and text editors, is not adequate for managing the complexity of most multicore embedded systems. Providing a development process with the help of a programming environment with facilities aimed at the development of these systems, represents a significant step towards reducing the drawbacks that make implementation one of the biggest bottlenecks during the design of these kinds of systems.

We have shown that with two existing tools, it is possible to develop multicore systems efficiently. The approach for the process of development is carried out by a combination of a graphical tool with a performance tool. By using a graphical tool it is possible to better understand the communication and synchronization of tasks, while the performance tool can tune the system. Our model starts by programming the application using the multitask concepts and then moving it automatically to run as a multicore application.

The advantage of using message in inter-core communication was shown. Shared memory creates shared state spaces and this make programming a difficult task since it is necessary to maintain the consistence while keeping control of the shared state on time. By using message, the problem is avoided because there are no longer inter core states, and therefore the states are local in relation to the cores.

By using a combination of RTOS multitask plus communication channels, we can easily move from multitask to multicore. Following our model, any application that can be modeled in any kind of RTOS tasks, can be implemented in multicore.

The possibility of quick experimentation makes the proposed development process quite adequate to create prototypes of multicore embedded application during the initial stages of development. By creating prototypes, problems existing in the design can be detected and fixed early, minimizing their consequences.

ACKNOWLEDGMENT

We would like to thank the Brazilian Agency **CNPq** (Conselho Nacional de Desenvolvimento Cientifico e Tecnológico) for the financial support, under grant no. **478084/2012-9**.

REFERENCES

[1] E. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, May 2006.

[2] A. S. Berger, *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*. CMP Books, Taylor & Francis, 2002.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 29–44, ACM, 2009.

[4] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs, "Your computer is already a distributed system. Why isn't your OS?," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2009.

[5] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the barrelfish manycore operating system," in *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[6] F. Nemati, R. Inam, T. Nolte, and M. Sjodin, "Towards resource sharing by message passing among real-time components on multi-cores," in *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pp. 1–4, Sept 2011.

[7] D. Pasetto, M. Meneghin, H. Franke, F. Petrini, and J. Xenidis, "Performance evaluation of interthread communicationmechanisms on multicore/multithreaded architectures," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, (New York, NY, USA), pp. 131–132, ACM, 2012.

[8] C. Clauss, S. Pickartz, S. Lankes, and T. Bemmerl, "Towards a multicore communications api implementation (mcapi) for the intel single-chip cloud computer (scc)," in *Parallel and Distributed Computing (ISPDC), 2012 11th International Symposium on*, pp. 148–155, June 2012.

[9] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," *SIGOPS Oper. Syst. Rev.*, vol. 13, pp. 3–19, Apr. 1979.

[10] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.

[11] J. Ribeiro, N. da Silva, R. Moron, and C. Moron, "From design to implementation using the parallel program generator," in *Euromicro Conference, 1998. Proceedings. 24th*, vol. 2, pp. 924–931 vol.2, Aug 1998.

[12] T. Delaitre, G. Ribeiro-Justo, F. Spies, and S. Winter, "A graphical toolset for simulation modelling of parallel systems.," *Parallel Comput.*, vol. 22, no. 13, pp. 1823–1836, 1997.

[13] A. Itzkovitz, A. Schuster, and L. Shalev, "Millipede: A user-level nt-based distributed shard memory system with thread migration and dynamic run-time optimization of memory references," in *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, NT'97, (Berkeley, CA, USA), pp. 19–19, USENIX Association, 1997.

[14] F. Heinze, L. Schafers, C. Scheidler, and W. Obeloer, "Trapper: eliminating performance bottlenecks in a parallel embedded application," *Concurrency, IEEE*, vol. 5, pp. 28–37, Jul 1997.

[15] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombás, "P-grade: A grid programming environment," *Journal of Grid Computing*, vol. 1, no. 2, pp. 171–197, 2003.

[16] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, pp. 287–311, May 2006.

[17] G. L. et. al., *IBM System Blue Gene Solution: Performance Analysis Tools*. Armonk, NY, USA: IBM Redpaper, 2014.

[18] V. Pillet, V. Pillet, J. Labarta, T. Cortes, T. Cortes, S. Girona, S. Girona, and D. D. D. Computadors, "Paraver: A tool to visualize and analyze parallel code," tech. rep., In WoTUG-18, 1995.

[19] J. Reinders, *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[20] P. F. Sweeney, M. Hauswirth, A. Diwan, M. Biberstein, and Y. Harel, "Understanding performance of multi-core systems using trace-based visualization," in *Proceedings of the First Workshop on Software Tools for Multi-Core Systems (STMCS06)*, March 2006.

[21] VisualRTXC, *Development Environment*. Houston, Texas, USA: Quadros Systems Inc., 2014.