

A Real-Time Programming Model for Heterogeneous MPSoCs

Torsten Limberg, Bastian Ristau, and Gerhard Fettweis

Technische Universität Dresden
Vodafone Chair Mobile Communications Systems
01062 Dresden, Germany
{limberg,ristau,fettweis}@ifn.et.tu-dresden.de

Abstract. Modern multi-core processors suffer from the lack of a programming model which allows efficient utilization of the available hardware. Massive software overhead is required to handle task scheduling and synchronization, resulting in power inefficiencies. In this paper we present a C++ based, real-time enabled task level programming model, which allows efficient hardware utilization. Task scheduling and synchronization is performed by a hardware unit at run-time. The automated scheduler unit is guided by offline information extracted from source code by a specialized compiler

1 Introduction

Heterogeneous MPSoC will be the preferred topology for high performance, low power signal processing systems of the future [1]. These systems will be set up from established, highly optimized components such as RISC processors, DSPs, ASIPs and ASICs. MPSoCs can deliver unlimited processing power to the programmer. Unfortunately programming such systems is not trivial [2]. Understandable programming models which utilize hardware effectively are required, in order to get the maximum benefit out of parallel systems.

Besides the programming model, MPSoCs impose other problems which could easily dilute the benefits drawn out of the parallelism. Two of the most profound problems are overheads introduced by context switches and hardware/software interfacing [3]. It is common practice to add specialized accelerators to signal processing systems in order to improve performance and power efficiency. However, the interface between hardware accelerators is usually interrupt based. This causes scheduling overhead for processing interrupt requests, drawing off compute power from the application and reducing the energy efficiency.

Another problem is the increasing number of applications running concurrently on modern signal processing systems such as mobile phones. Many of these applications are required to fulfill real-time requirements. This causes an increase of scheduling overhead which forces the hardware designers to put more powerful hardware into the system, causing further degradation of power efficiency. Future systems will have to cope with even more concurrent and dynamically changing applications. Unfortunately most existing programming models are totally

unsuitable for handling dynamics. This is mainly, because the user is required to perform resource allocation statically at compile time. With the increasing number of applications which are candidate to be run in parallel in a real-time environment, this is getting impractical. Therefore, dynamic resource allocation at run-time is required from our point of view.

We are addressing all of these challenges with our platform concept. To solve the problem of resource allocation and task synchronization we pick up the idea of having a task level programming model along with a dedicated hardware unit named CoreManager [4]. The CoreManager is absorbing a good portion of the operating system scheduler work and removes interrupts for accelerator synchronization completely. We extend [4] by adding real-time capabilities to the programming model and the CoreManager. According to [4], we call our real-time enabled CoreManager version Real-Time-CoreManager (RT-CoreManager). Offline information which is needed by the RT-CoreManager to perform task prioritization is added automatically by a specialized compiler.

In this paper we are focusing on the real-time programming model (section 4) and its required compiler (section 5). Nevertheless, a short introduction into hardware architecture is given in section 3. Section 6 gives some first results obtained from the compiler. A conclusion and an overview over future work can be found in section 7.

2 Related Work

In [5] recommendations for the development and evaluation of future “manycore” processors are given. MPSoCs are proposed to provide adequate processing capabilities at low power consumption. In order to program such architectures, more human centric, and efficient parallel programming models are claimed. Existing parallel programming models such as OpenMP [6], NVidias CUDA [7], Cilk [8] or μ TC [9] are targeted shared memory architectures. They extend the C language in order to allow the programmer to specify parallelism explicitly. However, these programming models exploit fine grain parallelism and heterogeneity is not supported. MPSoCs for application tailored signal processors require a programming model exploiting coarse grain parallelism, since fine grain instruction and data parallelism is already utilized by the single processing elements (e.g. with very long instruction words and single instruction multiple data processing).

Sequoia [10] is a programming model explicitly requiring an abstract description of the processor memory hierarchy. Programs are architecture independent and therefore portable to new targets. However, static mapping of tasks to resources is required.

A hardware concept for heterogeneous MPSoCs along with an appropriate C-based programming model is proposed by Seidel [4]. CellSs [11], which was developed to utilize the parallelism of the Cell BE [12], is closely related to this approach. Both propose a runtime scheduler which dynamically distributes tasks to processing elements. However, CellSs performs scheduling in software, while Seidels approach uses a dedicated hardware unit called CoreManager. We extend

Seidels work by adding real-time capabilities to the programming model and the hardware unit. When writing about real-time in this paper, we always refer to soft or semi-hard real-time constraints, since these are the types of constraints typically appearing in multimedia and communication systems.

3 MPSoC Platform Overview

A basic schematic of our MPSoC platform is depicted in Fig. 1. The software part is required in order to compile programs for the underlying hardware architecture. This will be detailed in section 5. In this section we will give a short overview over the hardware required to execute our real-time programs.

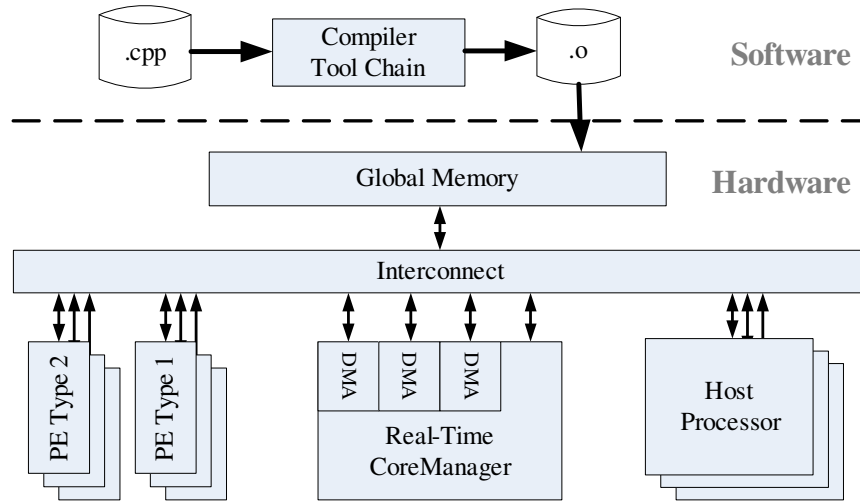


Fig. 1. Architecture schematic

Host Processor The system may have one or multiple host processors (HP) which are typically RISC processors. They are used to run the operating system and control code. The host processors share the global memory. Whenever a task is instantiated from control code, the HP sends a task description containing all relevant information for task execution to the RT-CoreManager.

Processing Elements A processing element (PE) can be any kind of hardware. This includes general purpose, domain and application specific processors as well as application specific fixed logic blocks or components for external interfacing. PEs are used to execute computational kernels. Each PE has its own local memory which is managed by the RT-CoreManager. PEs do not have direct access to external resources. In order to allow a wide range of existing components to be used, we do not expect PEs to be able to support context switching in hardware.

Real-Time CoreManager The RT-CoreManager is the key component of the system. It is responsible for mapping tasks to the processing elements and handles task prioritization and dependency checking. Furthermore it performs local memory management for the PEs and controls data transfers from and to them. The real-time extension prioritizes tasks based on the remaining time (slack) until the tasks latest possible start time weighted with the static priority of the thread. Scheduling is performed by a list scheduler which handles highest priority tasks first. In average 60 clock cycles are required to schedule one task.

Interfacing between RT-CoreManager and host processor is done without any interrupts. The HP sends task descriptions and receives status information over memory mapped registers.

4 Programming Model

The programming model is a crucial component of a MPSoC platform, since it influences the acceptance among programmers. Languages such as C, C++ or Java are popular and therefore more likely to be accepted than other languages. We use C++ as the base language in order to implement our programming model in the most convenient way.

Our real-time programs basically consist of two components: tasks and real-time threads. A task is an atomic computational kernel which is executed on a processing element. Tasks consume and produce chunks of data. They come in two different flavors: either a task is a program running on some kind of programmable processor or it is an algorithm executed on an ASIC. We assume tasks to have a run-time of at least a few hundred cycles.

Real-time threads are threads with an execution time limit. They live as normal threads in the real-time operating system and are executed interleaved on the host processor. Tasks are instantiated from the sequential thread code and are executed concurrently on the available processing elements. When a task was instantiated, the threads continues execution immediately. Thread execution only has to be stopped, if the task queue of the RT-CoreManager is full or if control code depends on data which is computed by one of the PEs. Synchronization statements to wait for data to be ready are generated automatically by our compiler. Synchronization between threads can be implemented using any mechanism provided by the used threading model.

Threads and Tasks have to be identified explicitly by the programmer. But in contrast to other approaches, the decision which kind of PE or even which concrete PE is used is not done at compile-time. Instead of this the programmer simply defines one or more types of PEs a task can be executed on and leaves the selection of the appropriate PE to the RT-CoreManager. For automatic extraction of task deadlines, the programmer annotates the maximum execution time of a task on each PE. The RT-CoreManager performs dependency checking between tasks at run-time and schedules a task not before all its dependencies are resolved. Thus, tasks are reordered at run-time. This is similar to instruction scheduling in superscalar processors.

4.1 Real-Time Support

Each thread has a static priority which reflects the importance of meeting its deadline. As an example threads of media applications will typically have less priority over communication applications in mobile phones. Static priorities and deadlines are assigned by the programmer.

In a real-time system only threads have an explicit deadline but task do not. To allow dynamic prioritization of tasks by the CoreManager, each tasks deadline in a thread must be annotated. Since this would be a cumbersome and error prone task for the programmer, we compute task deadlines using a specialized compiler. All information gathered by the compiler are annotated in the so called Thread Description Graph which is linked into program and guides the RT-CoreManager scheduling.

The Thread Description Graph (TDG) is a directed acyclic graph consisting of nodes representing task instances and edges representing possible control flow paths in the thread. Edges have annotated the probability of the control flow path to be taken. Tasks have annotated different parameters:

Program memory locations: For each possible implementation on a processing element, the CoreManager needs a pointer to the location of program code for this task. If the PE is an ASIC which does not require program memory, a unique dummy pointer is used. This is required since program memory pointers are used to track the path of execution in the RT-CoreManager. Starting from an initial root node provided with the start of a thread, the RT-CoreManager checks which path of execution is taken, whenever a new task arrives. The tracking mechanism compares the program memory pointers of the new task with the pointers annotated in the TDG to find out witch path was taken.

The **best case latest possible start time** (BCLPST) represents the latest start time of a task on a machine with unlimited PEs and DMA controllers under the assumption, that always the shortest control flow path is taken. The BCLPST is annotated in time units relative to the thread deadline. If a task is not able to meet its BCLPST, the corresponding thread is guaranteed to miss its deadline, too.

The **worst case latest possible start time** (WCLPST) is similar to the BCLPST but the longest control flow path is assumed for its calculation. Thus, when a task meets this deadline, it is guaranteed, that all succeeding tasks are also able to meet their deadline, too. However, if the WCLPST is missed, it is still possible to meet the thread deadline if a more optimistic case than that one assumed for WCLPST computation occurs. Therefore, we use the WCLPST for dynamic prioritization in the RT-CoreManager and the BCLPST as the cancellation criterion of a thread.

Loop information: When task calls are found in of loop bodies, special loop nodes are generated. Loop nodes denote start or end of a loop body. They have annotated their iteration count, the maximum execution time of the loop body as well as the deadline for the final iteration.

4.2 C++ Implementation

A thread is modeled by a class derived from `RtThread_t`. This base class has a pure virtual function `_Execute` which must be implemented by the programmer. `_Execute` contains the threads functionality. It must never be called directly from user code but is invoked implicitly when the threads `Start` function is called.

The `RtThread_t` base class encapsulates implementation details such as the threading model of the target operating system. Furthermore `RtThread_t` hides the details of communication with the RT-CoreManager from the programmer. Thus, user code does not need to contain any hardware or operating system specific information. Execution of code on another operating system or a new hardware with modified RT-CoreManager interface can easily be accomplished by simply replacing the `RtThread_t` base class.

Tasks are functions with a number of pointer arguments. They may be static class members or global non-member functions. Data is passed as pointer to its global memory location. Tasks are declared in the same way as normal C/C++ functions, but the definition is enclosed in some pragmas. These pragmas determine the entry point of the task and the processor types it may be executed on.

Listing 1.1 gives an example of a very basic real-time thread containing two task calls. For convenience the macros `TASK`, `IN` and `OUT` are available. `TASK` encapsulates a complete task call. The first parameter is the task name to be executed, the following parameters are the input and output data, which are passed to the task using the `IN` and `OUT` macros. Their arguments determine data location and the size of the data block.

Note that all extensions are implemented using standard C++ features, allowing compilation of the applications with any C++ compiler. Therefore, the program code can be used as reference as well as multi-core implementation without any changes. This allows functional verification to take place without any simulation runs on MPSoC simulators.

5 Compiler Tool Chain

For compilation, the C++ sources are passing a fully automated compiler tool chain. The complete chain is depicted in Fig. 2. In the first step, task and control code are separated by the TaskSplitter. Task code is compiled by the PE C/C++ compilers. The Thread Description Graph Compiler (TDGC) generates a C++ representation of the TDG for each thread in the application. The application source code along with the TDGs is compiled with the HP compiler. Finally the PE objects are linked as data blocks into the HP object code.

The TDG generated by the TDGC is fully annotated. Thus, no user modification is required. For TDGC implementation we extended the GCC source code in order support the special features of our programming model. Graph generation is performed in four successive steps. At first, control flow analysis is performed, followed by memory alias analysis. From the results of the first

two steps, TDGC computes the BCLPST and WCLPST as well as all data dependencies and annotates them in the graph. Finally the graph is transformed to reduce memory consumption.

During control flow analysis, a tree representation of the compiled thread is generated. In order to remove back-edges, we add loop-start and loop-end-nodes at the source and the destination of back-edges in the control flow graph. These loop-nodes contain the loop information and are required to make the graph acyclic.

```

// Task that can be executed on ASIP or DSP
#pragma TASK_BEGIN   FFT                // start task declaration
#pragma TASK_TARGET  FFT-ASIP,   410    // target, execution time
#pragma TASK_TARGET  StandardDSP, 2050  // target, execution time

// this is the task entry point
void FFT ( void *i1, void *i2, void *o ) {
    // function implementation goes here, even sub-function are allowed ...
}

#pragma TASK_END                // finish task declaration

// A very basic real-time thread class declaration
class MyThread_t :
    public RtThread_t {        // thread base-class

private:
    void _Execute ( );        // thread entry point function

    int    _mode;            // internal variables
    int*   _a, _b, _c;       // internal variables
    // further functions and variables ...
};

// Implementation of the thread functionality
void MyThread_t::_Execute ( ) {
    if ( _mode == 1 )        // if in the one mode, perform FFT ...
        TASK ( FFT, IN ( _a, 1024 * sizeof ( int ) ),
              IN ( _b, 512 * sizeof ( int ) ),
              OUT( _c, 1024 * sizeof ( int ) ) );
    else
        TASK ( IFFT, IN ( _a, 1024 * sizeof ( int ) ),
              IN ( _b, 512 * sizeof ( int ) ),
              OUT( _c, 1024 * sizeof ( int ) ) );
}

// main function implementation
int main ( ) {
    MyThread_t instance1;    // Create thread instances (but don't execute)

    // Start execution of the thread
    instance1 . Start ( 4, 10000 );    // priority=4, deadline in 10000 cycles

    // Start other threads here ...

    return 0;
}

```

Listing 1.1. C++ Implementation of a Real-Time Thread (The base class is `RtThread_t`, the thread entry function is `_Execute`)

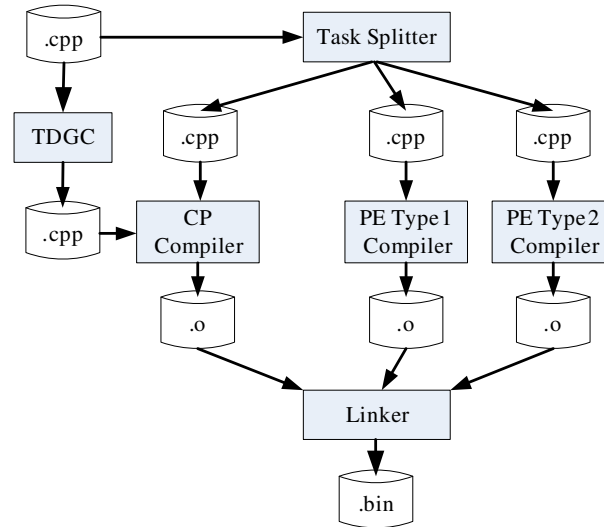


Fig. 2. The Compiler Tool Chain

During the memory alias analysis phase, task dependencies are analyzed. In order to do that, the compiler searches each control flow path separately for memory aliases [13]. A dependency is considered as present if at least one of the following conditions holds (assuming task B is issued after task A):

- One or more outputs of task A are aliased with one or more inputs of task B (read after write dependency).
- One or more outputs of task A are aliased with one or more outputs of task B (write after write dependency).
- One or more inputs of task A are aliased with one or more output of task B (write after read dependency).

If none of the above is true and the aliasing of inputs and output of two tasks is uncertain, the dependency between two tasks is annotated as uncertain. If no certain or uncertain dependency is found, two task are treated as independent.

The third compilation step computes the latest possible start times for each task node. For LPST computation we assume unlimited availability of PEs and DMA controllers. Network traffic is modeled according to the characteristics of the interconnection network. An "as late as possible" (ALAP) schedule is computed for this kind of machine, providing the start times for each thread in the TDG. The assumption of unlimited resources ensures, that the obtained deadlines are suitable for architectures with arbitrary parallelism, thus making the TDG only dependent on the kind of used PEs on an MPSoC but not on their count.

The last step tries to reduce memory consumption of the graph by merging nodes of equal tasks existing in different execution paths. In principle this could

be done for any nodes of equal tasks in the graph but we have to take care that BCLPST and WCLPST, do not diverge so much because of merging. In practice a restriction has to be placed on merging which limits this divergence.

6 Results

At the current development stage, the compiler is able to cope with arbitrary programs without loops. First tests on 802.11a WLAN C++ code resulted in 100% of correctly detected dependencies. Further tests with hand written test-cases resulted in 10% of uncertain task dependencies.

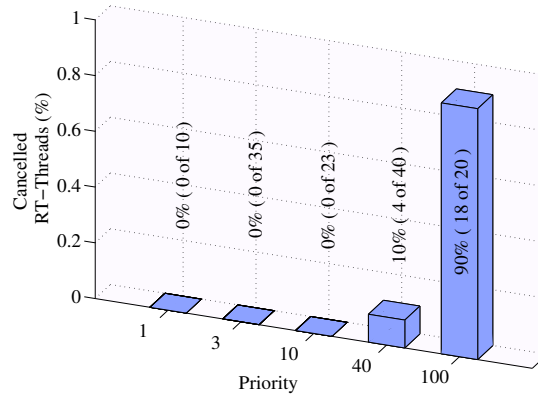


Fig. 3. Canceled tasks in an overloaded system

Figure 3 shows the cancellation rates of threads with different priorities which are executed in a heavily overloaded system. As it can be seen, only low priority threads (higher priority numbers denote less priority) are canceled due to the overload situation. The TDG for these threads has been generated with the TDGC. Results have been obtained by simulation on a transaction level simulator with 8 processors.

7 Conclusions and Future Work

In this paper we stated, that existing programming models and hardware architectures are not able to cope with future application requirements. We presented an alternative approach solving the problem of task synchronization and reducing scheduling overhead caused by hardware/software interfacing. Furthermore a real-time enabled programming model was suggested which makes concurrent real-time programming quite straightforward. Our first experiments showed, that a run-time scheduling is feasible under real-time constraints. Furthermore, it has

been shown, that the compiler can deliver results of the quality required for annotation of reasonable real time constraints.

Currently data flow analysis within loops is under development for the TDGC. Different real application benchmarks from the wireless communications and the multimedia domain are under development in order to prove the applicability of our approach to real time applications. Furthermore, we are aiming to allocate and manage communication resources in order to improve predictability of data transfer behavior.

References

1. Horowitz, M., Dally, W.: How scaling will change processor architecture. Proceedings of the IEEE Solid-State Circuits Conference, 2004, Digest of Technical Papers, ISSCC (February 2004) 132–133
2. Lee, E.A.: The problem with threads. *IEEE Computer* **39**(5) (May 2006) 33–42
3. Silven, O., Jyrkkä, K.: Observations on power-efficiency trends in mobile communication devices. *EURASIP Journal on Embedded Systems* **2007** (2007) Article ID 56976, 10 pages doi:10.1155/2007/56976.
4. Seidel, H.: A Task-level Programmable Processor. WiKu, Duisburg (October 2006)
5. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (Dec 2006)
6. Dagum, L., Menon, R.: Openmp: An industry-standard api for shared-memory programming. *IEEE Computational Science and Engineering* **05**(1) (1998) 46–55
7. Ghuloum, A., Sprangle, E., Fang, J.: NVidia: CUDA Programming Guide 1.1. http://www.nvidia.com/object/cuda_develop.html (Nov. 2007)
8. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.* **30**(8) (1995) 207–216
9. Bernard, T., Bousias, K., Geus, B.d., Lankamp, M., Zhang, L., Pimentel, A., Knijnenburg, P., Jesshope, C.: A microthreaded architecture and its compiler. (2006)
10. Fatahalian, K., Knight, T.J., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: Programming the memory hierarchy. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. (2006)
11. Bellens, P., Perez, J.M., M, R., Badia, Labarta, J.: CellSs: a programming model for the cell be architecture. Proceedings of the ACM/IEEE Supercomputing 2006 Conference (November 2006)
12. Pham, D., Asano, S., Bolliger, M., Day, M., Hofstee, H., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., Yazawa, K.: The design and implementation of a first-generation cell processor. Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 *IEEE International* **1** (February 2005) 184–592
13. Muchnick, S.S.: *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers, San Francisco (1997)