Single-Path Programming on a Chip-Multiprocessor System

Martin Schoeberl, Peter Puschner, and Raimund Kirner

Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at, {peter,raimund}@vmars.tuwien.ac.at

Abstract. In this paper we explore a time-predictable chip-multiprocessor (CMP) system based on single-path programming. To keep the timing constant, even in the case of shared memory access for the CMP cores, the tasks on the cores are synchronized with the time-sliced memory arbitration unit.

1 The Single-Path CMP System

The main goal of our approach is to build an architecture that provides a combination of good performance and high temporal predictability. We rely on chip-multiprocessing to achieve the performance goal and on an offline-planning approach to make our system predictable. The idea of the latter is to take as many control decisions as possible before the system is actually run. This reduces the number of branching decisions that need to be taken during system operation, which, in turn, causes a reduction of the number of possible action sequences with possibly different timings that need to be considered when planning respectively evaluating the system's timely operation.

1.1 System Overview

We consider a CMP architecture that hosts n processor cores, as shown in Figure 1. On each core the execution of simple tasks is scheduled statically as cyclic executive. All core's schedulers have the same major cycle that is synchronized to the shared memory arbiter. Each of the processors has a small local method cache (M$) for storing recently used methods, a local stack cache (S$), and a small local scratchpad memory (SPM) for storing temporary data. These caches store only thread local data and therefore no cache coherence protocol is needed. To avoid cache conflicts between the different cores our CMP system does not provide a shared cache. Instead, the cores of the time-predictable CMP system access the shared main memory via a TDMA bus with fine-grained statically-scheduled access.

1.2 Tasks

All tasks in our system are periodic. Tasks are considered to be simple tasks according to the *Simple-Tasks Model* introduced in [1]:[1] Task inputs are assumed to be available

[1] More complex task structures can be simulated by splitting tasks into sets of cooperating simple tasks.

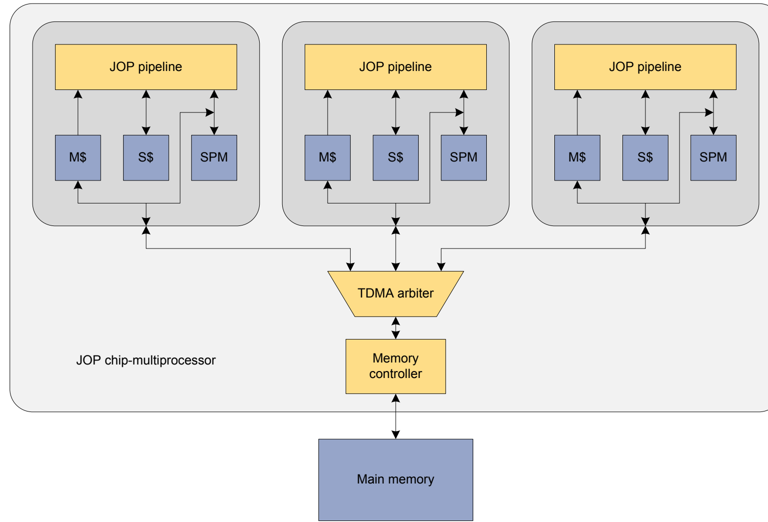

Fig. 1. A JOP based CMP system with core local caches and scratchpad memories, a TDMA based shared memory arbiter, and the memory controller.

when a task instance starts, and outputs become ready for further processing upon completion of a task execution. Within its body a task is purely functional, i.e., it does neither access common resources nor does it include delays or synchronization operations.

To realize the simple-task abstraction, a task implementation actually consists of a sequence of three parts: *read inputs – process – write outputs*. While the application programmer must provide the code for the *process* part (i.e., the functional part), the first and the third part are automatically generated from the description of the task interface. These *read* and *write* parts of the task implementations copy data between the local memories of the processing cores and the global memory. Care must be taken to schedule the data transfers between the local memories and the global memory such that all precedence constraints between tasks and mutual exclusion constraints are met.

Following our strategy to achieve predictability by minimizing the number of control decisions taken during runtime, all tasks are implemented in single path code. This means, we apply the single-path transformation described in [2, 3] to (a) serialize all input-dependent branches and (b) transform all loops with input-dependent termination into loops with a constant iteration count. In this way, each instance of a task executes the same sequence of instructions and has the same temporal access pattern to instructions and data.

1.3 Mechanisms for Performance and Time Predictability

By executing tasks on different cores with some local cache and scratchpad memory we manage to increase the system's performance over a single-processor system. The following mechanisms make the operation of our system highly predictable:

– Tasks on a single core are executed in a cyclic executive, avoiding cache influences due to preemption.
– Accesses to the global shared memory are arbitrated by a static TDMA memory arbitration scheme, thus leaving no room for unpredictable conflict resolution schemes and unknown memory access times.
– The starting point of all task periods and the starting point of the TDMA cycle for memory accesses are synchronized, and each task execution starts at a pre-defined offset within its period. Further, the single-path task implementation guarantees a unique trace of instruction and memory accesses. All these properties taken together allow for an exact prediction of instruction execution times and memory access times, thus making the overall task timing fully transparent and predictable.
– As the *read* and *write* sections of the tasks may need more than a single TDMA slot for transferring their data between the local and the global memory, *read* and *write* operations are pre-planned and executed in synchrony with the global execution cycle of all tasks.

Besides its support for predictability, our planning-based approach allows for the following optimizations of the TDMA schedules for global memory accesses. These optimizations are based on the knowledge available at the planning time:

– The single-path implementation of tasks allows us to exactly spot which parts of a task's *process* part need a higher and which parts need a lower bandwidth for accessing the global memory (e.g., a task does not have to fetch instructions from global memory while executing a method that it has just loaded into its local cache). This information can be used to adapt the memory access schedule to optimize the overall performance of memory accesses.
– A similar optimization is thinkable to optimize the timing of memory accesses during the *read* and *write* sections of the task implementations. These sections access shared data and should therefore run under mutual exclusion. Mutual exclusion is guaranteed by the static, table-driven execution regime of the system. Still, the critical sections should be kept short. The latter could be achieved by an adaption of the TDMA memory schedule that assigns additional time slots to tasks at times when they perform memory-transfer operations.

2 Implementation

The proposed design is evaluated in the context of the Java optimized processor (JOP) [4] based CMP system [5]. We have extended JOP with two instructions: a predicated move instruction for single-path programming in Java and a deadline instruction to synchronize application tasks with the TDMA based memory arbiter.

2.1 Processor Extensions

Single path programming substitutes control decisions (if-then-else) by predicated move instructions. To avoid execution time jitter, the predicated move has to have a constant

execution time. On JOP we have implemented a predicated move for integer values and references. This instruction represents a new, system specific Java virtual machine (JVM) bytecode. This new bytecode is mapped to a *native* function for access from Java code. The following listing shows usage of conditional move for integer and reference data types. The program will print 1 and true.

```
String a = "true";
String b = "false";
String result;
int val;

boolean cond = true;

val = Native.condMove(1, 2, cond);
System.out.println(val);
result = (String) Native.condMoveRef(a, b, cond);
System.out.println(result);
```

The representation of the conditional move as a native function call has no call overhead. The function is substituted by the system specific bytecode during link time (similar to function inlining).

In order to synchronize a task with the TDMA schedule a wait instruction with a resolution of single clock cycles is needed. We have implemented a deadline instruction as proposed in [6]. The deadline instruction stalls the processor pipeline until the desired time in clock cycles. We have implemented an I/O device for the cycle accurate delay. The time value for the absolute delay is written to the I/O device and the device delays the acknowledgment of the I/O operation until the cycle counter reaches this value. This simple device is independent of the processor and can be used in any architecture where an I/O request needs an acknowledgment. I/O devices on JOP are mapped to so called *hardware objects* [7]. A hardware object represents an I/O device as a plain Java object. Field read and write access are actual I/O register read and write accesses. The following code shows the usage of the deadline I/O device.

```
SysDevice sys = IOFactory.getFactory().getSysDevice();

int time = sys.cntInt;
time += 1000;
sys.deadLine = time;
```

The first instruction requests a reference to the system device hardware object. This object (sys) is accessed to read out the current value of the clock cycle counter. The deadline is set to 1000 cycles after the current time and the assignment sys.deadline = time writes the deadline time stamp into the I/O device and blocks until that time.

2.2 Evaluation

We evaluate our proposed system within a Cyclone EP1C12 FPGA that contains 3 processor cores and 1 MB of shared memory. The TDMA slot of each processor is 7 cycles and a complete TDMA round takes 21 cycles. As a first experiment we measure the

Table 1. Measured single-path execution time

Task	Read	Process	Write	Total
τ_1,τ_2	594	774	576	1944
τ_3	864	65250	576	66690
τ_4	26604	324	28422	55350
τ_5	1368	324	324	2016

execution time of a short program fragment with access to the main memory. Without synchronizing the task start with the TDMA arbiter the execution time of this single-path program is between 332 and 352 clock cycles. With the deadline instruction we can force that each iteration of the task starts at multiples of the TDMA round (21 clock cycles). In that case each task execution takes 336 cycles. This little experiment shows that single-path programming on a CMP system, synchronized with the TDMA based memory arbitration, results in *repeatable* execution time [8].

To validate our programming model for cycle-accurate real-time computing, we developed a controller application that consists of five communicating tasks. This case study give us some insights about the practical aspects of using the proposed programming model. Tasks τ_1-τ_5 are implemented in single-path code, thus their execution time does not depend on control-flow decisions. All tasks are synchronized on each activation with the same phase of the TDMA based memory arbiter. Therefore, their execution time does not have any jitter due to different phase alignments of the memory arbiter. With such an implementation style it is possible on the JOP to determine the WCET of each task directly by a single execution-time measurement. Table 1 shows the observed WCET values for each task, given separately for the read, process, and write part of the tasks.

3 Related Work

Time-predictable multi-threading is developed within the PRET project [6]. The access of the individual threads to the shared main memory is scheduled similar to our TDMA arbiter. The PRET architecture implements the deadline instruction to perform time based, instead of lock based, synchronization for access to shared data. In our approach we perform synchronization with three different execution phases.

The approach, which is closest related to our work, is presented in [9]. The proposed CMP system is also intended for tasks according to the simple task model [1]. Similar to our approach, a TDMA based memory arbitration is used. The paper deals with optimization of the TDMA schedule to reduce the WCET of the tasks. We think that this optimization can be best performed when the access pattern to the memory is statically known – which is only possible with single-path programming.

Optimization of the TDMA schedule of a CMP based real-time system has been proposed in [10]. The described system proposes a single core per thread to avoid the overhead of thread preemption. It is argued that future systems will contain many cores

and the limiting resource will be the memory bandwidth. Therefore, the memory access is scheduled instead of the processing time.

4 Conclusion

A static scheduled chip-multiprocessor system with single-path programming and a TDMA base memory arbitration delivers repeatable timing. The repeatable and predictable timing of the system simplifies the safety argument: measurement of the execution time can be used instead of WCET analysis. We have evaluated the idea in the context of a time-predictable Java chip-multiprocessor system. The cycle accurate measurements showed that the approach is sound.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 214373 (Artist Design) and 216682 (JEOPARD).

References

1. Kopetz, H.: Real-Time Systems. Kluwer Academic Publishers (1997)
2. Puschner, P., Burns, A.: Writing temporally predictable code. In: Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. (Jan. 2002) 85–91
3. Puschner, P.: Transforming execution-time boundable code into temporally predictable code. In Kleinjohann, B., Kim, K.K., Kleinjohann, L., Rettberg, A., eds.: Design and Analysis of Distributed Embedded Systems. Kluwer Academic Publishers (2002) 163–172 IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).
4. Schoeberl, M.: A Java processor architecture for embedded real-time systems. Journal of Systems Architecture **54/1–2** (2008) 265–286
5. Pitter, C., Schoeberl, M.: A real-time Java chip-multiprocessor. Trans. on Embedded Computing Sys. accepted for publication. (2009)
6. Lickly, B., Liu, I., Kim, S., Patel, H.D., Edwards, S.A., Lee, E.A.: Predictable programming on a precision timed architecture. In Altman, E.R., ed.: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008), Atlanta, GA, USA, ACM (October 2008) 137–146
7. Schoeberl, M., Korsholm, S., Thalinger, C., Ravn, A.P.: Hardware objects for Java. In: Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008), Orlando, Florida, USA, IEEE Computer Society (May 2008)
8. Lee, E.A.: Computing needs time. Commun. ACM **52**(5) (2009) 70–79
9. Rosen, J., Andrei, A., Eles, P., Peng, Z.: Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In: Proceedings of the Real-Time Systems Symposium (RTSS 2007). (Dec. 2007) 49–60
10. Schoeberl, M., Puschner, P.: Is chip-multiprocessing the end of real-time scheduling? In: Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis, Dublin, Ireland, OCG (July 2009)