# Code Generation and Context Switching for Static Scheduling of Mixed Control and Data Oriented HW/SW Systems

Achim Österling, Thomas Benner, Rolf Ernst

Technical University of Braunschweig
Institut für Datenverarbeitungsanlagen
Hans–Sommer–Str. 66
38106 Braunschweig
Germany

## Abstract

*Code generation and hardware and software scheduling are important topics in hardware/software co–design. Different context switching mechanisms are used depending on the type of application and the underlying hardware. In particular in systems combining control and periodic data processing functions context switching overhead can be significant. So, the switching effects should also be regarded in hardware/software co–synthesis. The three context switching mechanisms for non–preemptive scheduling, pico kernel, code merging, and table based sequencing, which shall represent the span of known techniques, are investigated. The results show a large impact on code size and performance. They demonstrate that in mixed applications code merging and table based sequencing can be much more efficient than the pico kernel approach which has a context switching overhead similar to a real–time kernel. The paper discusses some of the reasons. The results suggest to include context switch selection in the co–design process. We finally propose a technique to improve code merging and table based sequencing.*

## 1 Introduction

Despite many improvements in co–synthesis during the past years, the range of applications and target architectures is still rather limited and prevents design space exploration for more complex architectures. There is work in reactive, control dominated systems, such as [3, 6, 7, 8] and work in data dominated or computation oriented systems, such as [1, 9, 10, 11] which differ in the type of time constraints and data operations which they can handle as well as in the optimization process.

Many applications, such as mobile communications, however, show a mixture of both control and data dominated applications on different levels of a design, often included in a single process, such as a wireless communication channel. Data dominated applications, in particular periodic applications in signal processing and control engineering applications, make extensive use of buffering to increase the design space and to allow high performance pipelining on various levels of granularity (statements up to processes). On the other hand, the current control dominated co-synthesis approaches which focus on relatively small systems with processor-coprocessor architectures make little or no use of buffering and pipelining.

A major problem of mixed systems are the latency constraints of reactive processes which conflict with data dominated system optimization. Preemptive scheduling using interrupts is problematic since it disturbs pipeline flow and exhibits long context switching times in the order of a few microseconds.

In [4], a process model is defined which is based on non–preemptive scheduling. The results show its effectiveness but the effectiveness of such a non–preemptive approach is obtained at the cost of short context switching intervals. This requires a fast yet low memory overhead context switch (memory is critical in embedded systems). In this paper, we investigate several context switching mechanisms.

In the following section, we will review related work in co–synthesis to give a better impression of code generation in current co–synthesis approaches. Section 3 presents several code generation candidates. Then, we present results, give an outlook on improvements, and draw some conclusions.

## 2 Related work

In this section we present related work in co–synthesis. This section is divided into two parts. First, we present systems for reactive applications. In the second part, systems for data dominated applications are presented. Finally, we show what problems must be considered when allowing applications with reactive and data dominated parts at the same time (mixed systems).

In [6], the POLIS co–design environment is presented. As input, the user supplies a description in ESTEREL which is translated to CFSMs. Alternatively, the CFSMs may be specified directly. There is one code segment per FSM transition. Since the code segments corresponding to the FSM states are short there is no need for an intermediate context switch. The target architecture consists of one microprocessor and several hardware modules. The hardware part is translated to a hardware description language. Code generation is based on logic synthesis. For the software part, a Real–Time OS is needed. This OS consists of a scheduler and drivers for I/O channels. The scheduling algorithm can be selected by the user, e.g. Rate–Monotonic and Deadline–Monotonic. Hardware synthesis creates one hardware module per CFSM with point–to–point communication between hardware modules while hardware modules and processor communicate via shared registers.

TOSCA [3] assumes a target architecture similar to the one in POLIS, however, user interaction, the partitioning algorithm, and the input languages are different. Again, there is one hardware module per FSM and point–to–point communication between FSMs while hardware/software communication uses driver templates and shared registers or memory.

The approach of [7] differs in input language (LOTOS), partitioning algorithm, and synthesis. The target architecture consist of a microprocessor and an ASIC. Communication is performed via shared memory. The scheduling method is non–preemptive round–robin.

The Chinook co–design environment, presented in [8], differs from the co–synthesis systems described above. Here, the input is an annotated Verilog description. Chinook already considers larger segments of code which need to be interrupted to meet I/O constraints. More specifically, there is a set of global system control states, called modes, which correspond to these code segments. Switching between these segments could be compared to threads in event chains. Each thread is scheduled statically and can be left at predefined exit points, thereby performing the mode transition. This requires a context switch. Target architecture is a processor with hardware modules, one hardware module corresponding to one Verilog process. Communication is performed via signal assignments (Verilog). Blocking can be implemented by using wait statements. The approach in [12] has another thread concept splitting HardwareC processes into linear segments separated by I/O and communication operations. Their target architecture comprises a single processor and an ASIC. Communication is performed via a bus.

Co–synthesis systems for data–dominated applications typically use non–preemptive scheduling, e.g. [1, 9, 10, 11]. [1] has a fixed blocking communication scheme using message passing between hardware modules and a processor. One statically scheduled process cluster is mapped to the processor whereas the hardware modules contain a number of concurrent process clusters. A single global time constraint (suitable for periodic systems) is supported and guides partitioning. In [10], the SpecSyn environment is presented. There, the partitioning algorithm focuses on the mapping of behavior to system components, the mapping of variables to memory, and the mapping of communication to buses. Partitioning is mainly directed by closeness metrics, i.e. behavior blocks are grouped if they are close to each other w.r.t. certain metrics. The same holds for memory and bus assignment. In contrast to [10], [9] uses shared memory communication.

Even though data dominated applications have high computation requirements, it is not necessary to interrupt a function execution since I/O latency (deadline) constraints are not the dominant problem. So, context switching may be coarse grained which simplifies code generation. A typical approach is the encapsulation of code segments in functions which simplifies code generation and supports hierarchy (see e.g. [13]). This is unlike [8] where loops need to be unrolled to meet deadline constraints. Obviously, if we move on to mixed systems, we need to split function execution in order to meet I/O constraints, just like in [8] or [12].

The system representation in [11] is similar to that in [12]. A concurrent process description is translated to a constraint graph consisting of a number of threads. Each thread may be separated from each other by I/O and communication operations (unbounded delays). Threads are clustered such that a cluster is invoked on arrival of an event. An event may be a completion of an unbounded operation or an internal event, i.e. the event is raised when all predecessors of a cluster have finished execution. There are two execution model options. First, a number of thread clusters can be invoked by processor interrupts. Moreover, thread clusters can be suspended by assigning priorities to the interupts. Another option is a preemptive time–driven run–time scheduler that invokes the thread clusters based on a remaining time heuristic.

However, preemptive context switching is limited to software implementations and, thus, constrains the hardware/software design space.

We conclude that for mixed systems large code segments must be split in small intervals. A potential problem may be loops, and a solution would be to unroll them. However, this may be too expensive in general. Furthermore, data dominated systems can have complex contexts, such as stacks and buffers. So, keeping many concurrent contexts can be costly in terms of memory. This is a problem for preemptive context switching, but also for non–preemptive scheduling if we allow, e.g., ”splitting” in deeply nested functions.

In the following, we investigate several context switching techniques and discuss their cost, performance, and generality.

# 3 Context switching techniques

The three techniques which we investigate are

- Pico kernel
- Code merging
- Table based sequencing

## 3.1 Pico kernel

The pico kernel is a small process which is used for context switching by saving the context of the current process and activating the context of the process which is scheduled to execute next (fig. 1a). This is done by simply inserting a call to the context switch function at the position the scheduler determined for a context switch (*schedule point*).

The context switching time is predictable, because the pico kernel does not include code with data dependent execution times. However, context switching is time consuming. In case of a SPARC processor and the GNU C compiler, we measured about 100 clock cycles (3 $\mu s$) using the ”longjump” function. Also, in co–design, the pico kernel allocation can become a problem since a context switch destination can be in hardware or in software. If the pico kernel is placed in software, each context switch requires processor interaction, such that the size of a hardware function in co–synthesis is bounded to code segments between two schedule points. If the pico kernel is allocated to hardware, context switching becomes hardware–controlled. This would constrain the optimization potential and would lead to increased communication overhead when scheduling granularity decreases. Thus, in general, there will be a copy of the pico kernel in each component. As a caveat, context switching must be balanced to avoid stack overflow.

## 3.2 Code merging

Code merging combines the scheduled process segments into an inlined code sequence. Fig. 1b shows the principle. This way the compiler is used to optimize context handling and minimize context switching time and data memory overhead for stack frames. Code merging also supports the compiler which can now optimize across context switches with a positive influence on code motion and register assignment.

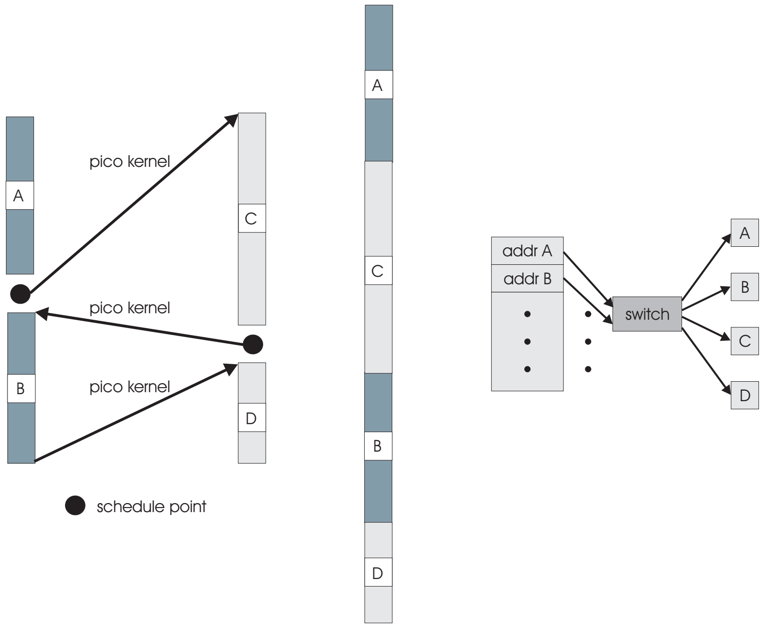
A critical question is where to allow the context switch. When we allow a process switch in a conditional structure we have to inline the process for each branch. This can result in code explosion and, therefore, does not seem feasible in general.

The option we chose requires that all schedule points be placed on the outermost scope. This implies that blocking communication cannot be allowed in conditional structures. If required to meet timing constraints, inner schedule points have to be moved outside by partial loop unrolling and restructuring.

Assume, e.g., a for–loop is interrupted during the n$^{th}$ iteration. The first part contains the iterations 1 to n-1, the second one contains the n$^{th}$ iteration including the schedule point, and the third part consists of the remaining iterations. The results are three different structures derived from a single loop. So, the advantage of a faster context switch might be paid with a much larger code size. This intuitive conclusion is not true in general, but depends on the architecture, the application, and the scheduling algorithm, as we will see in the results.

## 3.3 Table based sequencing

This technique tries to combine the advantages of the pico kernel and of code merging. By generating a sequencer to control con-




(a) pico kernel (b) code merging (c) table based sequencing

**Figure 1: Code generation techniques**

text switching, code duplication can be avoided. The sequencer is encoded in a switch statement.

For scheduling, all code segments are encapsulated in functions and called by a software FSM. This software FSM is encoded in a switch statement (fig. 1c). Compiler optimization is more constrained compared to code merging, but still provides more optimization potential then the pico kernel approach which prevents global optimization.

## 4 Results

In this section we compare the three methods described above, the pico kernel, code merging, and table based sequencing. To evaluate context switch efficiency, code generation must be combined with a non–preemptive scheduler. We used the SPS scheduler [5] for the experiments. The target architecture is a single processor. Process communication uses send and receive functions.

In order to investigate how the code generation techniques affect the required speedup and the code size, we tried variants of two examples. The first one is a model train control. The train is controlled by a personal computer, from which 35 bit values are transferred via the rails. The model train has a fairly powerful motion speed regulation that moves the train close to original large trains. A decoder scans the pulse duration on the rails. Due to noise on the rails, a low pass filter and a 3 bit burst error correction code are used. The corrected and decoded value is passed to a regulator which receives the actual velocity from a speedometer. Finally, a motor controller generates pulses for the motor driver.

A part of an ethernet bridge specification [2] serves as a second example. Its function is to transmit data packages from one segment to another. Thereby it learns the network structure in order to minimize the netload. When the bridge receives a packet from a segment, it tries to find the address in a database. If the address is found, it sends the packet to a specific segment. Otherwise, the packet is sent to all segments. Fig. 2 shows the code size of the processes. For all processes, periods are defined.

| benchmark | process | period | code size (Byte) |
|---|---|---|---|
| train buffered/ non–buffered | decoder | $t_1$ | 288 |
| | receiver | $16 \cdot t_1$ | 1628 |
| | regulator | $16 \cdot t_1$ | 688 |
| | pdm | $16 \cdot t_1$ | 248 |
| | rpm | $16 \cdot t_1$ | 148 |
| total | | | 3000 |
| bridge | d22 | $t_3$ | 428 |
| | d23 | $2 \cdot t_3$ | 544 |
| | d24 | $2 \cdot t_3$ | 260 |
| | d25 | $2 \cdot t_3$ | 164 |
| | d32 | $2 \cdot t_3$ | 484 |
| | d33 | $2 \cdot t_3$ | 508 |
| | d34 | $2 \cdot t_3$ | 254 |
| total | | | 2642 |

**Figure 2: Periods and code sizes of the original processes**

Given the time constraints, the output of a real–time scheduler and code generator is a feasible schedule or an indication of the non–feasibility of a schedule. Such a qualitative result is not useful for quantitative comparison of context switching and code generation. Instead, we kept the relative periods of the processes constant and tried to minimize the execution time of all processes. In the train example, when the period of process decoder is $t$, the period of process rpm is always $16 \cdot t$. Fig. 2 compares the relative periods and the code sizes of the processes.

For the train benchmark, SPS generated two different schedules. The first one assumes an architecture which only buffers a single input event. In this case, the processes have to be activated in time. In the second case input events are buffered and the processes can be delayed. This is useful when the deadline of a process is larger than

| benchmark | technique | $t_i$ ($\mu s$) | code size (byte) |
|---|---|---|---|
| Train buffered | PK | $t_1$ = 11.5 | 3376 |
| Train buffered | CM | $t_1$ = 3.1 | 4552 |
| Train buffered | TBS | $t_1$ = 5.0 | 3800 |
| Train non-buffered | PK | $t_2$ = 20.9 | 3376 |
| Train non-buffered | CM | $t_2$ = 5.9 | 8888 |
| Train non-buffered | TBS | $t_2$ = 10.0 | 3800 |
| Bridge buffered | PK | $t_3$ = 30.1 | 3344 |
| Bridge buffered | CM | $t_3$ = 22.7 | 3136 |
| Bridge buffered | TBS | $t_3$ = 22.8 | 3144 |

**Figure 3: Code size of the scheduled processes after code generation**

the period. In this case, the scheduler can arrange the processes to reach a better processor utilization. For the train example, buffering leads to a much smaller execution time. Since the order of the processes of the bridge does not change when buffering is used, the impact of the code generation technique is not as large as that of the train example.

Fig. 3 compares the impact of the different code generation techniques on the benchmarks, measured on a SPARC 20/50. For each example, the effects of the three code generation techniques are compared, the pico kernel (PK), code merging (CM), and table based sequencing (TBS). The code overhead is smaller for the pico kernel than for code matching (12% versus 52%), but code merging produces extremely fast results. For the buffered train the decoder process can be executed with a period of 3.1 $\mu s$ without hardware acceleration. TBS turns out to be a good compromise.

The number of process switches depends on the ratio of the process periods. While the ratio of the periods is 2 w.r.t. process d22 in the case of the bridge example, process decoder of the train example is executed 16 times as often as the other processes.

The number of process switches per macro period is an important factor for the differences in execution time. That is why the techniques influence the train example much more than the bridge example. The execution time is maximal for the pico kernel because of the two required context switches per process switch. The code merging technique avoids the context switches and allows global compiler optimizations. In the case of the train benchmarks, CM leads to a lower required speedup. TBS embeds segments of the code into a switch statement.

Due to the small differences of the bridge process's periods, the effect of the code generation techniques is smaller for this example. Interestingly, the code sizes hardly differ for this example with a clear speed advantage for TBS.

# 5 Extensions to the context switching approaches

In this section an extension to the context switching approaches presented in section 3 is presented. With this extension it is possible to allow schedule points to be located in conditional structures.

In case of a conditional branch, the extended scheduling algorithm, which shall be called *conditional code motion*, regards the longest path. If a process switch is scheduled inside the branch (fig. 4a), a corresponding code position in an alternative branch at the same time is selected to balance the number of context switches on all paths. This is shown in fig. 4b. Now, the conditional branch is split into two branches at the schedule points, i.e. the conditional structure is closed before a schedule point and re–opened after it, thus moving the schedule point out of the conditional structure.

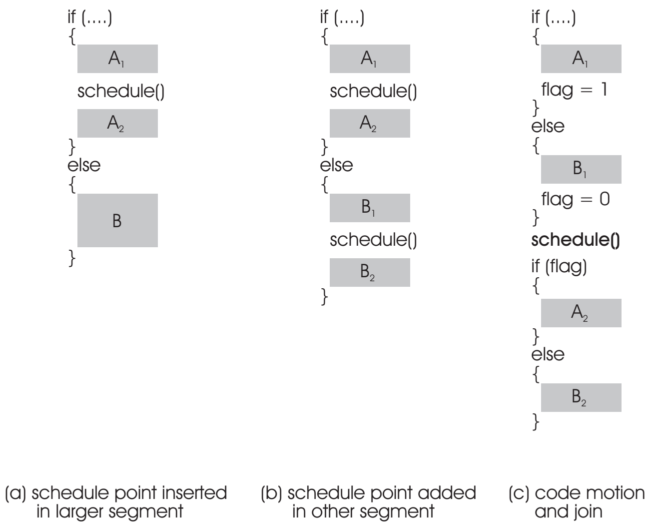



**Figure 4: Conditional code motion**

In conjunction with partial loop unrolling, applying these methods step by step for each nest of the program, the schedule points will eventually have been moved to the top level.

# 6 Conclusion

We presented a comparison of three code generation techniques for non–preemptive scheduling in the context of hardware/software co–synthesis of mixed systems. The effects of the techniques on the required system speedup and the code size were determined. Code size and performance vary widely depending on the application. The compiler optimization potential and the selected schedule have a large impact on context switching efficiency.

Moreover, we saw that event buffering does not only reduce execution time but also code size. We conclude that the selection of code generation should be included in the co–synthesis process.

# References

[1] J.K. Adams, D.E. Thomas, *Multiple–Process Behavioral Synthesis for Mixed Hardware/Software Systems*, Proc. of 8th Intentional Symposium on System Synthesis, Cannes, France, Sep. 13–15, 1995.

[2] F. Backes, *Transparent bridges for interconnection of IEEE 802 LANs*, IEEE Network, Vol. 2, No. 1, 1988.

[3] A. Balboni, W. Fornaciari, D. Sciuto, *Cosynthesis and Co–Simulation of Control–Dominated Embedded Systems*, Design Automation for Embedded Systems, Vol. 1, No. 3, Kluwer, 1996.

[4] T. Benner, R. Ernst, *A Combined Partitioning and Scheduling Algorithm for Heterogeneous Multiprocessor Systems*, Technical Report CY–96–2, Institut für DV–Anlagen, Technische Universität Braunschweig, 1996.

[5] T. Benner, R. Ernst and A. Österling, *Scalable Performance Scheduling for Hardware/Software Cosynthesis*, Proc. of the European Design Automation Conference, Brighton, Sep. 18–22, 1995.

[6] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, A. Sangiovanni–Vincentelli, *A Case Study in Computer–Aided Co–design of Embedded Controllers*, Design Automation for Embedded Systems, Vol. 1, No. 1–2, Kluwer, 1996.

[7] C. Carreras et. al., *A Co–Design Methodology Based on formal Specification and High–Level Estimation*, IEEE Proc. of Fourth International Workshop on Hardware/Software Codesign, Pittsburgh, Pennsylvania, 1996.

[8] Pai Chou, Ross B. Ortega, Gaetano Borriello, *The Chinook Hardware/Software Co–Synthesis System*, International Symposium on System Synthesis, Cannes, France, Sep. 13–15, 1995.

[9] R. Ernst, J. Henkel et al., *The COSYMA environment for hardware/software cosynthesis of small embedded systems*, Microprocessors and Microsystems 20 (1996), Elsevier Science B.V., 1996

[10] D.D. Gajski, F. Vahid, S. Narayan, J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, Englewood Cliffs, NJ, USA, 1994.

[11] F. Thoen, M. Cornero, G. Goosens, H. De Man, *Real–Time Multi–Tasking in Software Synthesis for Information Processing Systems*, International Symposium on System Synthesis, Cannes, France, Sep. 13–15, 1995.

[12] R.K. Gupta, G. De Micheli, *Constrained Software Generation for Hardware–Software Systems*, Proc. of 3rd Int'l Workshop on Hardware/Software Codesign, Grenoble, France, Sep. 22–24, 1994.

[13] F. Vahid, *Procedure Cloning: A Transformation for Improved System–Level Functional Partitioning*, Proc. of European Design and Test Conference, Paris, France, Mar. 17–20, 1997.

[14] W. Ye, R. Ernst, T. Benner, J. Henkel, *Fast Timing Analysis for Hardware–Software Co–Synthesis*, Proc. of ICCD 1993, Cambridge, pp. 452–457, 1993.