COOPERATIVE MULTITHREADING IN DYNAMICALLY RECONFIGURABLE SYSTEMS

Enno Lübbers and Marco Platzner

University of Paderborn
email: {enno.luebbers, platzner}@upb.de

ABSTRACT

Preemptive multitasking, a popular technique for time-sharing of computational resources in software-based systems, faces considerable difficulties when applied to partially reconfigurable hardware. In this paper, we propose a cooperative scheduling technique for reconfigurable hardware threads as a feasible compromise between computational efficiency and implementation complexity. We have implemented this mechanism for the multithreaded reconfigurable operating system ReconOS and evaluated its overheads and performance on a prototype.

1. INTRODUCTION

Multithreaded programming is an increasingly popular way to express concurrency in applications. It allows to decompose an application into separate threads of execution, which can be synchronized using a defined set of programming model objects provided by an operating system. With the concurrency made explicit through the usage of threads, such an application can be mapped to a parallel execution environment, for example a multiprocessor machine. However, if there are more threads than processing elements, a way has to be found to share the computing resources.

Conventional software operating systems employ several multitasking techniques. *Non-preemptive* techniques are simple to implement and simplify thread synchronization on uniprocessor systems, albeit at the cost of unpredictable latencies and no support for asynchronous operations. At the other end of the spectrum, *preemptive* multitasking enables the perceived simultaneous execution of multiple tasks on a single processor with low latencies, some level of predictability, and support for asynchronous (i.e. blocking) functions; it does, however, require elaborate synchronization schemes and incurs a moderate scheduling overhead. For a certain set of applications, there exists a middle ground in *cooperative* multitasking. This technique enables individual threads to voluntary relinquish control of the processor to allow other threads to execute. With appropriately 'behaved' tasks, it can reduce latencies compared to non-preemptive methods, while, on uniprocessor systems, avoiding many of the synchronization pitfalls of preemptive multitasking. In software-based systems, though preemptive multitasking seems to be the more popular method for implementing multithreaded applications, cooperative techniques are still used within GUI toolkits and for the implementation of language features such as Python's *generator* functions or Perl's *yield* statement. In fact, Moura et. al. [1] advocate the increased use of coroutines, which in essence are cooperatively executing functions, as a general control abstraction in modern programming languages.

In the context of multithreaded systems for reconfigurable hardware, selecting an appropriate multitasking technique becomes a more complex challenge. Kalte et. al. [2] implement a mechanism using bitstream readback to extract and later re-inject the contents of storage elements known to contain a task's state information. While flexible and transparent, this technique involves considerable overhead, since the contents of CLB registers can make up less than 1% of an FPGA's configuration bitstream. An alternative approach is the inclusion of register scan chains in a task's logic, as proposed by Jovanovic et. al. [3]. It does, however, incur a significant area and time overhead when preempting a task.

In this paper, we propose using cooperative multitasking techniques together with partial reconfiguration for sharing FPGA logic resources between hardware modules as a feasible compromise between execution efficiency and implementation complexity. We have implemented cooperative multitasking for hardware threads in the ReconOS [4] environment, effectively combining a preemptive scheduling technique for software threads on the system's CPU with cooperative scheduling for hardware threads in the FPGA's fabric. To evaluate our method's overheads and demonstrate its feasibility, we have performed measurements on a prototype FPGA implementation.

2. RECONOS

This section gives an overview of the programming model and execution environment which make up the ReconOS operating system for CPU/FPGA platforms. A more detailed discussion can be found in [5] and [4].

The ReconOS project extends the multithreaded programming model, as implemented in many contemporary soft-

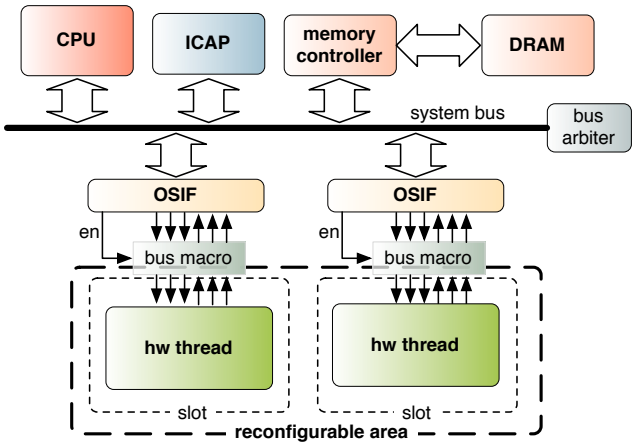

Fig. 1. ReconOS system architecture

ware operating systems, to the domain of reconfigurable hardware. Instead of modelling hardware components mapped to an FPGA's logic fabric as passive coprocessors, ReconOS treats hardware modules as independent threads. To enable hardware threads to access OS objects, such as semaphores or mutexes, the corresponding software API calls are exposed to the hardware through a VHDL library. The procedures provided by this library can be used in a state machine description to manipulate defined OS interface signals, which in turn evoke the appropriate control functions in a separate operating system interface module (OSIF). The OSIF then either executes the requested function, if it can be handled in hardware, or relays the OS call to the CPU by means of an interrupt line and software-accessible registers. On the CPU, a software thread called the *delegate* then executes the actual OS call using the operating system's native API. Thus, a ReconOS hardware thread typically consists of a OS synchronization state machine and an arbitrary number of concurrent "user logic" VHDL processes which carry out the thread's intended functionality.

When implemented in Xilinx FPGAs, ReconOS uses dynamic partial reconfiguration via the ICAP interface to replace hardware threads during run-time. The dynamically reconfigurable region is divided into a fixed number of *slots*, each of which is associated with an OSIF, as depicted in Figure 1. All signals between the OSIF and a hardware thread inside a slot are routed through bus macros.

3. HARDWARE MULTITASKING TECHNIQUES

The unique properties of a hardware thread implemented in an FPGA make it difficult to apply traditional multitasking schemes to enable resource-sharing between different threads of computation. A general-purpose microprocessor has a fixed set of registers, the contents of which define the state of the computational entity currently executing. This makes it simple to suspend a computation and save its state – it is sufficient to transfer the register contents of the processor to memory. There is no need to store any information about the actual processing units, since they are identical for all computations performed on the device and cannot be modified. Also, the code saving the state is usually executed on the same processor as the suspended computation, and thus has direct access to the state information.

In contrast, a computation mapped onto an FPGA typically does not have an easily identifiable location defining its state. Instead, the state may be distributed across different storage elements – flip-flops, shift registers, embedded memories – throughout the FPGA's fabric. In addition, not all storage elements may be used by a given configuration; indeed, not even all used storage elements may contribute to a computation's state. To make matters worse, there is no easy way to store the contents of a given computation's storage elements into memory, since the logic connected to the storage elements is dedicated to the computation itself.

3.1. Non-preemptive Multitasking

A simple solution is not to interrupt threads during their execution at all, but letting them run from creation to termination. ReconOS supports non-preemptive multitasking for hardware threads by providing `create()` and `exit()` calls for hardware threads. When a hardware thread is created, its delegate thread requests the configuration of the appropriate bitstream to a free slot, if available. In the same way, upon termination of a hardware thread, its delegate thread marks the slot as free, which allows another thread waiting for a slot to use it. Let us consider a simple example of two hardware threads A and B with respective execution times of $T(A)$ and $T(B)$ and a reconfiguration time of t_l for either thread. When run on a single slot using non-preemptive multitasking, the total run time is $T_n(A, B) = 2t_l + T(A) + T(B)$.

A significant disadvantage is that long-running subroutines prevent other pending events from being handled, thereby making the system appear non-responsive. Also, asynchronous (i.e. blocking) operations – I/O, for example – should not be invoked directly but be registered with the event loop by providing a callback function, in order not to delay other events unneccessarily. This breaks up the logical structure of most applications. For these reasons, software systems often use this technique in conjunction with preemptive multitasking, which again has the aforementioned drawbacks for reconfigurable hardware.

3.2. Cooperative Multitasking

Seen in this light, cooperative multitasking techniques, while appearing unnecessarily handicapped in software-based systems given the easy availability of preemptive multitasking, present an acceptable compromise when applied to reconfig-

urable hardware. Most of the problems described in Section 1 stem from the fact that in preemptive systems, a thread can be interrupted at arbitrary points during its execution. If we leave the actual decision to suspend a thread to the thread itself, it can select *what* state information to store, since the location and amount of that information is thread-specific, decide *how* to store the state by providing access logic only for the relevant state information, and choose *when* to suspend itself, sensibly picking a point in time when state information is minimal. The last point is especially valuable for threads which frequently wait for data packages to process – during the (typically blocking) wait periods, such a thread often has minimal state information to be saved.

The logic requirements for the hardware thread in cooperative multitasking environments, while easier to implement than mechanisms for preemptive multitasking, are still higher than those for simple non-preemptive multitasking. At the same time, cooperative multitasking offers better utilisation of the reconfigurable resources than non-preemptive techniques, provided that the threads actually relinquish control (i.e. *yield*) of their computational resources.

A thread typically signals its readiness to be suspended by calling a `yield()` function. If no other threads are waiting for a slot, this call would return immediately with no consequences for the thread. Ideally, this is done just before periods in which the thread would not perform any computation anyway. An excellent point for relinquishing control are therefore blocking operating system calls.

In our previous example of threads A and B, we can imagine that A actually does not use the whole period $T(A)$ for computations, but calls a function which blocks for t_{block} until it returns. During this time, A *yields* the slot to other waiting threads (in this case, B) after saving its state. This allows the operating system to remove A, reconfigure the waiting B into the (now free) slot, and execute it. After B terminates and marks the slot as free, A is reconfigured, its state restored, and its execution resumed. The total run-time of both threads then amounts to $T_c(A, B) = 2t_l + T(A) + t_{A_s} + t_{A_r} + max(t_{block}, t_l + T(B))$.

t_{A_s} and t_{A_r} are the times it takes to save and restore A's state, respectively. The cooperative multitasking approach reduces the total run-time, provided that both $T(B) > t_{A_s} + t_{A_r}$ and $t_{block} > t_{A_s} + t_{A_r} + t_l$.

3.3. Implementation

In ReconOS, the cooperative scheduling technique is only employed for hardware threads contending for slots on the FPGA. Software threads running on the CPU still use preemptive scheduling, since it is comparatively cheap to implement in software. The task of managing the reconfigurable resources in ReconOS is shared between a hardware thread's *delegate thread* and a high-priority software thread, the *hardware scheduler*, both of which are managed by the OS' preemptive scheduler. This approach provides excellent portability, since it does not involve any changes to the underlying software operating system.

We refer to the synthesized hardware circuit representing a specific thread's functionality as a *core*. For every available *slot* in the system, a hardware core is placed and routed, resulting in $n_{slots} \times n_{cores}$ partial *bitstreams*. There can be multiple hardware *thread* instantiations based on the same core. The data structures modeling the relationships between slots, hardware threads, cores and bitstreams are shared between the delegates and the hardware scheduler.

Upon invocation, the hardware scheduler checks all hardware threads' and slots' scheduling information. For any hardware thread waiting for execution, it looks for a slot – free or possibly with a yielding thread in it – to reconfigure it into. In the event that there are no free slots, the scheduler sends a request to yield to all running threads. This mechanism allows hardware threads to easily check for other hardware threads waiting to be executed, avoiding unnecessary saving of state information. If a hardware thread is newly started, its delegate requests its associated hardware core to be loaded by setting a flag in its scheduling data structure and notifying the hardware scheduler. After the corresponding partial bitstream has been configured to the FPGA, the delegate continues waiting for and serving OS call requests from the hardware. If any of these calls carries a 'yielding' flag set by the hardware, the thread notes this in its scheduling data structure and notifies the scheduler. Upon completion of such a call, this flag is again cleared. Should the thread's hardware core have been replaced during the call, the delegate thread requests the reloading of its core and wakes up the scheduler. After reconfiguration, the hardware thread's state is restored and normal execution resumes.

4. EXPERIMENTS

To evaluate the cooperative scheduling technique, we have measured the timing overheads of individual operations as well as total application execution time of a prototype implementation. The prototypes employs a ReconOS/eCos operating system on a XC2VP30 FPGA platform using a single reconfigurable slot and an operating frequency of 300 MHz and 100 MHz for the CPU and the reconfigurable logic, respectively.

Table 1 shows the execution times of individual operations that can be invoked by the hardware thread, as well the overheads of thread control, state save/restore and reconfiguration operations performed by the operating system. The results show that the overhead incurred by the scheduling API remains acceptably small, particularly in the event that no other hardware thread is waiting for a slot. Thus, augmenting a hardware thread with cooperative scheduling operations will not have a significant impact on the perfor-

Table 1. Execution times of thread and OS operations

thread operation	cycles
`check_yield()`	0
`flag_yield()`	0
`thread_yield()` (no thread waiting)	4
`thread_yield()` (threads waiting)	700
`thread_resume()`	4
store/load state to/from local RAM (32 bit)	1
store state directly to main memory (32 bit)	26
load state directly from main memory (32 bit)	32
OS operation	
thread initialization	1.76 ms
thread suspend	93.12 μs
thread resume	192.32 μs
state save (4096 bytes)	37.51 μs (104.1 MB/s)
state restore (4096 bytes)	45.19 μs (86.4 MB/s)
reconfiguration time (233 kBytes)	99,96 ms

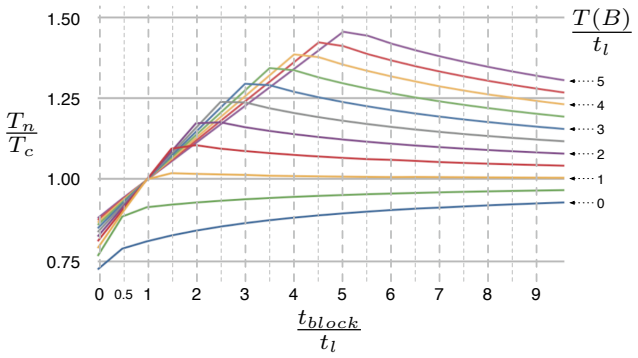

Fig. 2. Cooperative vs. non-preemptive multitasking

mance of the thread itself. However, the reconfiguration process is by far the most expensive operation. For the reconfigurable regions (13 $\times$ 17 CLBs) used in our prototype, which require partial bitstreams of 233 kBytes each, the reconfiguration time dwarfs any memory accesses for state saving or restoring. While this time can be reduced by more than an order of magnitude by improving the configuration port interface core [6] or by minimizing the bitstream size [6][7], it remains a limiting factor that determines what applications can be feasibly implemented with any multitasking technique. We have run a range of benchmarks using the two threads A and B from our previous example. $T(A)$ has been fixed it at 65.5 ms, while $T(B)$ and t_{block} have been varied from 0 to 500 ms / 1000 ms, respectively. The resulting performance gains compared to the non-preemptive technique are shown in Figure 2. As long as either $T(B)$ or t_{block} are shorter than the reconfiguration time, cooperative multitasking executes slower than the sequential non-preemptive techinique due to reconfiguration and scheduling overhead. If both parameters rise above 100 ms, the performance is determined by the relation between $T(B)$ and t_{block} and peaks when $T(B) = t_{block}$.

5. CONCLUSION

In this paper, we have shown cooperative multitasking techniques to be a feasible compromise between computational efficiency and implementation complexity when sharing reconfigurable resources among hardware threads. We have extended the multithreaded programming model provided by ReconOS to support cooperative scheduling techniques, demonstrated its feasibility on a prototype and evaluated the overheads incurred by the scheduling operations.

A cooperatively scheduled system can offer the benefit of reduced synchronization complexity, as the threads cannot be preempted in a critical area. This advantage is in part neutralized in a truly concurrent execution environment which again requires explicit synchronization to avoid race conditions. As part of our ongoing work, we are investigating how to best integrate a cooperatively scheduled subset of threads in a concurrently executing multithreaded system. Additionally, we are working on integrating existing improvements on the reconfiguration infrastructure in order to implement more complex case studies that take advantage of our multitasking environment.

This work was supported by the German Research Foundation under project number PL 471/2-2.

6. REFERENCES

[1] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 2, pp. 1–31, 2009.

[2] H. Kalte and M. Porrmann, "Context Saving and Restoring for Multitasking in Reconfiugrable Systems," in *15th IEEE Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2005, pp. 223–228.

[3] S. Jovanovic, C. Tanougast, and S. Weber, "A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems," in *2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2007, pp. 358–364.

[4] E. Luebbers and M. Platzner, "Multithreaded Programming for Reconfigurable Computers," *ACM Trans. Embedded Computing Systems (TECS)*, 2009, to appear.

[5] ——, "ReconOS: An RTOS supporting Hard- and Software Threads," in *17th IEEE Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2007.

[6] C. Claus, F. Muller, J. Zeppenfeld, and W. Stechele, "A new framework to accelerate Virtex-II Pro dynamic partial self-reconfiguration," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007, pp. 1–7.

[7] M. Rullmann, S. Siegel, and R. Merker, "Optimization of reconfiguration overhead by algorithmic transformations and hardware matching," in *19th Int. IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2005, pp. 151–156.