Hybrid Prototyping of Multicore Embedded Systems

Ehsan Saboori and Samar Abdi
Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada
{e_saboo, samar}@ece.concordia.ca

Abstract— **This paper presents a novel modeling technique for multicore embedded systems, called *Hybrid Prototyping*. The fundamental idea is to simulate a design with multiple cores by creating an emulation kernel in software on top of a single physical instance of the core. The emulation kernel switches between tasks mapped to different cores and manages the logical simulation times of the individual cores. As a result, we can achieve fast and cycle-accurate simulation of symmetric multicore designs, thereby overcoming the accuracy concerns of virtual prototyping and the scalability issues of physical prototyping. Our experiments with industrial multicore designs show that the simulation time with hybrid prototyping grows only linearly with the number of cores and the inter-core communication traffic, while providing 100% cycle accuracy.**

Keywords-Embedded systems; Validation; Multicore design; Virtual prototyping, FPGA prototyping

I. INTRODUCTION

We present a new technique called Hybrid Prototyping that offers the scalability benefits of virtual prototypes, as well as the cycle-accuracy of FPGA prototypes. The fundamental idea of hybrid prototyping is to create a multicore emulation kernel (MEK) in software that executes on a single target core that is physically implemented in FPGA. The MEK simulates the execution of concurrent tasks on independent cores by dynamically scheduling the tasks on the physical target core. The MEK manages the state of the individual cores and the logical simulation times. The contributions of this work are (i) the hybrid prototyping methodology and (ii) the design of the multicore emulation kernel.

Most virtual platform technologies are based on binary translation, as commercialized by Windriver [1], Coware[2], and Xilinx XVP[3], where instruction-set simulation has replaced or complemented traditional cycle-accurate micro-architecture simulators [4-7]. Such simulators can provide significant speedups (reaching simulation speeds of several hundred MIPS), but often focus on functionality and speed at the expense of limited or no timing accuracy. Host-compiled software simulation techniques are based on source level timing annotation [8, 9]. The delays are derived by analyzing the application execution on an abstract model of the core, which leads to estimation inaccuracies. RAMP and Protoflex platforms uses an FPGA array to support the instantiation and integration of hundreds of cores [10, 11]. Unfortunately, the cost and design time of such full system prototypes is very high. In addition, there is no flexibility of abstracting the inter-core communication in RAMP, since it is fixed in hardware by the inter-FPGA communication architecture.

978-3-9815370-0-0/DATE13/©2013 EDAA
This work was funded by the *Fonds québécois de la recherche sur la nature et les technologies* (FQRNT) new researcher grant no. 146497.

Our technique of hybrid prototyping is distinct from the above approaches in that it time-multiplexes several virtual cores on a single physical target core that can easily fit on a single FPGA chip. Since the application executes on exactly the same core as it is targeted for, the estimation accuracy is 100%. Furthermore, there is no need for availability of source code or knowledge of the core datapath, since the application binary runs directly on the target core.

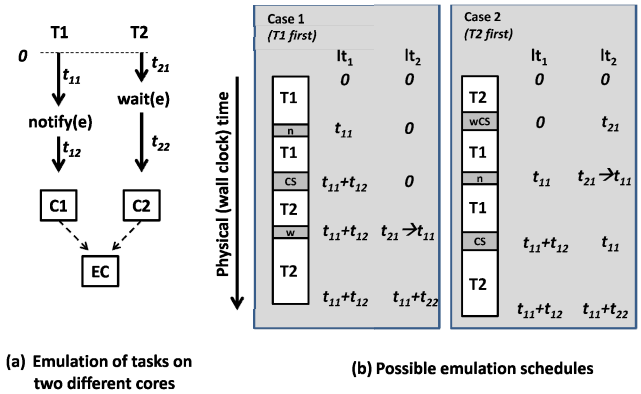

Figure 1. Simple example of simulation with MEK

II. METHODOLOGY

Figure 1 uses a simple example to illustrate multi-core simulation on a hybrid prototype. The design consists of multiple cores, communicating using simplex channels. The synchronization in the channels between tasks on different cores is modeled using events. We assume a discrete event model, where an event is consumed by a waiting task, or is lost if no task is waiting at the logical time of notification.

Figure 1(a) shows the design with two cores, each executing a single task. Task T1 executes on core C1 for time t_{11} and notifies a global event *e*. After notification, it executes for another t_{12} units and terminates. Task T2 executes on core C2 (of the same type as C1) for time t_{21} ($< t_{11}$) and waits for the global event e. After e is notified (by T1), it executes for another t_{22} units and terminates. Both tasks are assumed to start at the same time. The cores, C1 and C2, are simulated by a single core *EC*, which is of the same type as C1 and C2, and hosts the MEK.

Figure 1(b) shows two possible simulation schedules on EC. A task may be in four possible states: RUNNING, READY, BLOCKED or TERMINATED. The MEK maintains the logical times, lt_1 and lt_2, on C1 and C2, respectively. The logical time for a core is the time until which the core has been simulated. At logical time 0, the

MEK may pick either C1 or C2 to simulate first. If the MEK schedules C1 to be simulated first, it runs T1 on EC until *e* is notified. The MEK saves the event's notification and its logical timestamp t_{11}. Since event notification is non-blocking in a discrete event model, the MEK allows T1 to execute until it is terminated. Then, the MEK does a context switch (CS) and runs T2 from its logical time 0 until it reaches *wait(e)* at logical time t_{21}. At this point the MEK checks for any notifications of e that were made after logical time t*21*. Indeed, since $t_{11}>t_{21}$, the MEK finds that *e* was notified by T1 before T2 executed *wait(e)*. Therefore, the MEK updates the logical time of C2 to t_{11} to model T2 being blocked on the wait from t_{21} to t_{11}. Finally, T2 is resumed and runs to completion.

If the MEK schedules C2 to be simulated first (Case 2), it runs T2 on EC from C2's logical time 0 until it reaches *wait(e)* at C2's logical time t_{21}. Since no notifications of e are found, the MEK stores the wait on *e* with timestamp t_{21}, and blocks T2. It then does a context switch from C2 to C1. To emulate C1, the MEK runs T1 from C1's logical time 0 until the notification of e at C1's logical time t_{11}. Upon notification, the MEK checks if there are any pending waits on *e* at or before logical time t_{11}. Indeed, task T2 is blocked since C2's logical time t_{21} ($< t_{11}$) on e. Therefore, the MEK unblocks T2 and updates C2's logical time to t_{11} in order to account for the blocking time. The MEK continues simulating C1 until termination of T1, followed by a context switch to C2 and its simulation until termination of T2.

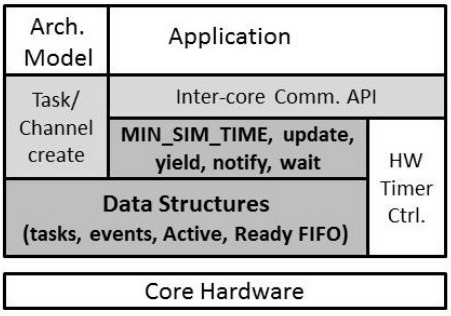

Figure 2. The layered MEK structure for hybrid prototyping

III. MULTICORE EMULATION KERNEL

Figure 2 shows the MEK structure in grey. The lowest layer is the MEK data structure, consisting of tasks, events and the scheduling FIFO queue that keeps all the READY tasks. A special global pointer *Active* is used to identify the running task. The next higher layer consists of the simulation primitives for the management of events and logical times of the tasks. The models of the communication channels are implemented as an API on top of the simulation primitives. The architecture is modeled using task creation and channel creation methods as shown. The application uses the inter-task communication API provided by the MEK.

Since, hybrid prototyping depends on measurement of physical time, a hardware timer and its drivers are included. The MEK uses a hardware timer to measure the execution time in CPU cycles. It resets and starts the timer before the block by calling *TimerStart()*. At the end of the block, the MEK calls *TimerStop()* to stop the timer and reads the timer's value by calling *TimerVal()*.

A. Data Structures

The MEK has two fundamental structures: *task* and *event*. The task contains (i) the *status* of the task (running, ready, blocked or terminated), (ii) the logical *time* until which it has been simulated, (iii) the *entry function* in the application code corresponding to the task, and (iv) the task context, consisting of the *program counter* (where the task must resume), the *stack pointer* and the *CPU register* values. The event structure consists of two lists *notifylist* and *waitlist*. The item type in each list is a pair of task pointer and timestamp. As the name suggests, *notifylist* is the list of all tasks that have notified the event and the corresponding logical time of notification. Similarly, *waitlist* is the list of all tasks that are waiting on the event, and the corresponding logical time when the wait was initiated.

In addition to the tasks and events, the MEK keeps a pointer to the currently running task called *Active*. The simulation scheduling policy used in the MEK is First-In-First-Out (FIFO). Therefore, the MEK also keeps a FIFO queue of pointers to tasks in the ready state. During simulation scheduling, tasks may be en-queued at the end of the FIFO by calling *putFIFO* method and the task at the top of the FIFO may be de-queued using *getFIFO* method.

B. Simulation Primitives

The MEK supports context switching between tasks (cores) during simulation is done by the *context_switch* method that saves the context (program, stack pointers and registers) of the previous task and loads the context of the new *Active* task. The MEK also maintains a global variable called MIN_SIM_TIME, which keeps the minimum logical time until which all tasks (cores) have been simulated.

The MEK provides an *update* primitive, which sets a given task's *time* field to a given time. The method recalculates MIN_SIM_TIME and removes all event notifications that were made before MIN_SIM_TIME, since discrete event semantics dictate that event notifications without a waiting task are lost. The MEK also provides an *yield* primitive, that can only be called by the *Active* task to allow other cores to be emulated. The primitive changes the caller's status to READY and reinserts the caller into the scheduler FIFO. It selects the new *Active* task from the head of the scheduler FIFO and switches the context.

```
void notify (event e)
1: if (∃ w ∈ e.waitlist, w→timestamp ≤ Active→time) {
2:         w→task→status = READY;
3:         putFIFO (w→task);
4:         delete (e.waitlist, w);
5: }
6: else
7:         add (e.notifylist, Active, Active→time);
```

Listing 3: Notify

Listing 3 shows the pseudo code for notification of given event *e*. The method looks for a task that had called a *wait(e)* at a logical time before the current logical time of the

notifying *Active* task (line 1). If such a wait is found, the status of the waiting task is change to READY (line 2). The task is inserted into the scheduler queue and the wait is deleted (lines 3-4). If no waiting task is found, it is possible that the task which might call the wait at an earlier logical time has not yet been emulated till the wait call. Therefore, the notification is added to the *notifylist* of *e*.

```
void wait (event e)
1: if (∃ n ∈ e.notifylist, n→timestamp ≥ Active→time)
2:         delete (e→notifylist, n);
3: else {
4:         task *t = Active;
5:         add (e→waitlist, t, t→time);
6:         t→status = BLOCKED;
7:         Active = getFIFO();
8:         Active→status = RUNNING;
9:         context_switch (t, Active);
10:}
```

Listing 4: Wait

Listing 4 illustrates the pseudo code for a *wait* on event *e*. The method looks for an event notification at a logical time later than the current logical time of the *Active* task calling the wait. A notifying task may have been simulated for a longer time that the waiting task and the notification may be present in the *notifylist*. If the notification is found, it is removed from the event's *notifylist* and the caller proceeds (lines 1-2). If a notification is not found, we must allow other tasks to run, so that a potential notify on the event is executed. The wait is added to the event's *waitlist* and the waiting task is BLOCKED by the MEK (lines 4-6). The MEK reschedules by obtaining the new *Active* task from the scheduler queue and scwitching the context (lines 7-9).

C. Channel Communication Model

The basic simulation primitives of notify, wait, update and yield are powerful enough to build complex communication models. In this section, we will describe modeling of simplex channels for point-to-point communication between the cores, as a representative example. The channel is implemented as a circular buffer:

```
struct { // circular buffer
    void *items[SIZE];
    int read_time[SIZE], write_time[SIZE];
    int head, tail;
    bool empty, full;
    events ev_read, ev_write;
} typedef channel;
```

The channel buffer is modeled as an *array of items* of user defined type and size. A read and write timestamp is associated with each item. The head and tail of the circular buffer is maintained as well. Readers read from the head and writers write into the tail. The channel has boolean variables to indicate a *full* or *empty* state, as well as respective events that are notified whenever the buffer is read or written.

Listing 5 illustrates the pseudo code for a blocking write (*bwrite*) into the channel *ch*. The timer is stopped to mark the end of user code and the start of the communication model (line 1). The timer value is used to update the logical time of the *Active* caller task (line 2).

```
void bwrite (channel *ch, void *data)
1:  TimerStop();
2:  update(Active, Active→time + TimerVal());
3:  while (ch→full) {
4:      if (Active→time == MIN_SIM_TIME)
5:          wait (ch→ev_read);
6:      else
7:          yield ();
8:  }
9:  if (ch→rd_time[ch→tail] > Active→time)
10:         update (Active, ch→rd_time[ch→tail]);
11: // write data and update flags/time...
12: ch→wr_time[ch→tail] = Active→time;
13: notify (ch→ev_write);
14: TimerStart();
```

Listing 5: Blocking write

As per blocking semantics, the writer must wait if the channel is full. If the logical time of the Active task is same as MIN_SIM_TIME, then all other tasks (including the reader of this channel) have been simulated at least until this time. Therefore, the writer must block on the channel read event (lines 4-5). Otherwise, it is possible that the reader may not have been simulated until the current logical time of the writer. Hence, the writer yields for the reader to be simulated until its current logical time (lines 6-7). If the current tail of the circular buffer (where the writer will write the new data) was read at a logical time after the writer's current logical time, it implies that the buffer was full at the time of the attempted write. As such, the writer would block until the tail item is read. We account for the blocking time in such a scenario by updating the writer's current time to the read timestamp on the tail (lines 9-10). The actual writing is subsequently done by copying over the data into the buffer's tail, updating the buffer full flag, if needed, and incrementing the writer's logical time with the time it takes to write into the channel (line 11). Finally, the logical time of completing the write into the tail is recorded, the write event is notified, and the timer is started before the method returns (lines 12-14). The blocking read method is a dual of the write method.

IV. Experimental Results

To evaluate the speed and accuracy of hybrid prototypes, we used a JPEG encoder application, which consists of 5 tasks, where each task consumes a frame of image data, processes it and passes the block to the next task. The application can be pipelined and the concurrent tasks can be mapped to different cores. We chose the Microblaze core from Xilinx for the target multicore architectures [3]. The FIFO communication between the tasks is performed using the Fast Simplex Link (FSL) buses supported by Microblaze, which is a circular buffer channel that implements the blocking protocol, as described in the model in Section IV-C.

We created both the physical FPGA prototype and the hybrid prototype for 16 multicore designs of the JPEG encoder, ranging from 1 core to 5 cores. For each platform, we used different mappings from tasks to cores. All the

Microblaze cores used were clocked at 100 MHz. Each Microblaze core in the physical prototypes has 64KB of dedicated Block RAM (BRAM) for program and data. The hybrid prototypes used a single Microblaze core with 64KB BRAM, since all the tasks and the MEK fit into a single BRAM. For the discussion below, design *Ni*, refers to the i^{th} mapping with *N* cores.

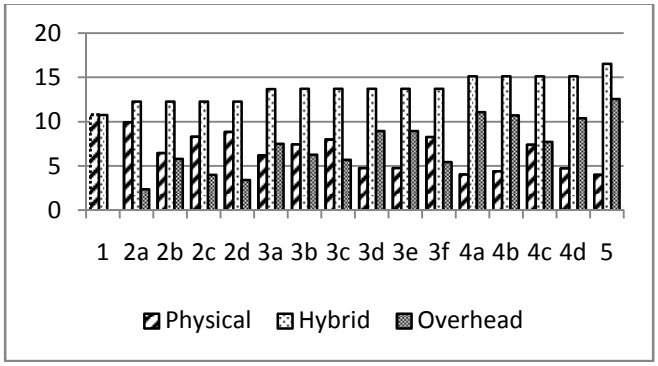

Figure 4. The simulation times for physical and hybrid design

Figure 4 shows speed comparison between hybrid and physical prototypes using the number of cycles needed to execute the JPEG encoder for a given image. The X-axis is the design and the Y-axis is the number of cycles in millions. The *overhead* is defined as the difference between the number of cycles needed for simulating JPEG on the hybrid prototype, and those for executing JPEG on the physical prototype. As we can see, the hybrid prototype takes approximately the same time for all mappings with a given number of cores. This is because the total inter-core data communication is the same for different mappings of JPEG. The small variations are due to different absolute communication times for each channel, and the variations in task scheduling in the MEK. We can also see that the worst case overhead for a given number of cores (in designs 1, 2b, 3d, 4a, 5) scales well with the number of cores and the amount of data communication.

The overhead of the MEK itself can be observed as the difference between the hybrid prototype times and the 1-core JPEG physical prototype execution time, since the total computation on the core stays constant. The MEK overhead consists of the wall clock time used for task/event management, scheduling and channel calls. As we can see, the MEK overhead also scales well with the number of cores and the amount of channel communication.

In the most complex design with 5 cores, the hybrid prototype took 16.5 K cycles (or 165 ms) to simulate JPEG. On the other hand, the physical prototype took 4K cycles (or 40 ms). In contrast, the behavioral RTL simulation of the 5-core design took over 3 hours on a 2GHz Pentium host with 8GB of RAM. We were unable to create a 5-core virtual prototype, because the Xilinx Virtual Platform (XVP) simulator supports only a single instance of Microblaze [3]. For the 1-core design, XVP took 3 minutes to simulate JPEG on the same host as the one used for RTL simulation. Based on the above results, we can conclude that hybrid prototyping outperforms both cycle-accurate RTL software simulation and virtual prototypes.

The hybrid prototype reported exactly the same number of cycles for each task as measured by the physical prototype. This is because we execute the tasks on the same core as in the physical prototype. The idle times for each core are accounted for by the accurate simulation of blocking time during channel read/write. Finally, we also account for the time to read/write data from/to the channels. In contrast, the XVP simulation had an error of over 50% in the number of cycles reported because of the high abstraction level of the underlying ISS. Therefore, our hybrid prototype was more accurate than abstract virtual prototypes.

V. CONCLUSION

In this paper we have presented a new modeling technique called hybrid prototyping that aims to provide early, fast, cycle-accurate and scalable models of multicore embedded systems. Using hybrid prototypes, embedded software designers can create concurrent applications and accurately analyze the performance implication of their optimizations before the hardware is available. Multicore architects can optimize the hardware architecture without having to do full system prototyping. Therefore, hybrid prototypes can provide huge productivity gains for both embedded software designers and multicore chip architects. In the future, we will extend the hybrid prototyping approach to support cores running at different frequencies, complex inter-core communication architectures, such as shared buses and NoCs, as well as memory hierarchies.

REFERENCES

1. Wind River Simics. Available at http://www.windriver.com/products/simics/
2. Coware Platform Studio. Available at http://www.synopsys.com/Tools/SLD/
3. Xilinx Embedded Development Kit. Available at http://www.xilinx.com/edk
4. F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX*, 2005.
5. M. Reshadi, P. Mishra, and N. Dutt, "Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation," *ACM TECS*, vol. 8, no. 3, 2009.
6. G. Braun, et al., "A universal technique for fast and flexible instruction-set architecture simulation," *IEEE TCAD*, vol. 23, no. 12, pp. 1625–1639, 2004.
7. W. S. Mong and J. Zhu, "DynamoSim: a trace-based dynamically compiled instruction set simulator," in *ICCAD* 2004.
8. Y. Hwang, S. Abdi, and D. Gajski, "Cycle approximate retargettable performance estimation at the transaction level," in *DATE*, Munich, Germany, Mar. 2008.
9. Z. Wang and A. Herkersdorf, "An efficient approach for system-level timing simulation of compiler-optimized embedded software," in *DAC*, San Francisco, CA, Jul. 2009.
10. Wawrzynek, J.; et al. , "RAMP: Research Accelerator for Multiple Processors," *Micro, IEEE* , 27(2), pp.46-57, 2007
11. Chung, E., et al., "ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs," in *ACM Trans. On Reconfigurable Technology Syst.* 2(2), pp 1-32, 2009