**D3.1**

| | |
|---|---|
| Revision: | 1.0 |
| Date: | 2011-08-31 |
| Author: | UPB |
| Dissemination Status: | PU |
| Document Reference: | D3.1 |

Project no. 257906

**EPiCS**

**Engineering Proprioception in Computing Systems**

# Autonomous Heterogeneous Multi-Core Architecture and Basic Tool Flow

Due date of deliverable: 2011-08-31 (T0+12)

Dissemination Level: **PU** (Public)

Lead partner for this deliverable: UPB

Contributing partners: EADS, ETHZ

| | | |
|---|---|---|
| Project acronym: | EPiCS | |
| Project name: | Engineering Proprioception in Computing Systems | |
| Call and Contract: | FP7-ICT-2009-5 | |
| Grant agreement no.: | 257906 | |
| Project duration: | 2010-09-01 – 2014-08-31 (48 months) | |
| Coordinator: | UPB | University of Paderborn (DE) |
| Partners: | IMPERIAL | Imperial College London (UK) |
| | UIO | University of Oslo (NO) |
| | UNI-KLU | University of Klagenfurt (AT) |
| | UOBIRM | University of Birmingham (UK) |
| | EADS | EADS Innovation Works (DE) |
| | ETHZ | ETH Zürich (CH) |
| | AIT | Austrian Institute of Technology (AT) |

This project is supported by funding from the FET proactive initiative "Self-Awareness in Autonomic Systems" by the European Union 7th Framework Programme.

## Document History

| Revision | Date | Scope of changes | Description | Implemented by |
|---|---|---|---|---|
| 0.1 | 2011-08-09 | All sections | New document | UPB |
| 0.2 | 2011-08-27 | All sections | Prepared draft | UPB |
| 1.0 | 2011-08-31 | All sections | Revised and released | UPB |

# Contents

## 1 Executive Summary

This deliverable report gives an overview of the work that has been done in Task T3.1 during the months 4 to 12 of the EPiCS project. We have developed an autonomous heterogeneous multi-core architecture based on the hardware/software operating system ReconOS. We have devised and implemented a monitoring infrastructure that allows us to collect information about the system's internal state, an important precondition for creating a self-aware compute node. In accordance with the description of work this report also documents the tool chain that is used to build and deploy the proposed multi-core system as well as two reference designs that serve as an entry point for working with and developing applications for heterogeneous compute nodes. All custom software and scripts needed to build our system (except commercial vendor-specific tools) as well as the reference designs presented in this report can be obtained from the EPiCS ReconOS repository (tag 'epics-reference-v1.0') at `http://github.com/EPiCS/reconos`.

## 2 Introduction

From the description of work for Task T3.1:

> "In this task, we will set up and implement the heterogeneous multi-core architecture for a single autonomous compute node as specified in WP 1. The multi-core will be implemented on a modern platform FPGA and comprise dedicated and soft CPU cores, reconfigurable hardware cores, and the monitoring core for capturing the internal system state and thus enabling self-awareness. The cores will be attached to a shared bus system to enable a shared memory programming model which is appropriate for the targeted systems.
>
> We will develop the basic tool flow for the heterogeneous multi-core including compilation tools and standard operating system support. To this end, we will adapt and integrate as much as possible existing software (C/C++) and hardware (VHDL) tool chains for the different cores. As operating systems, we will support the open source eCos for real-time embedded computing , and the open-source Linux operating system for high-performance embedded and general purpose computing. The resulting tool flow will utilise vendor tools to lay out a complete node architecture, to compile a collection of software threads for the operating system, to synthesise hardware functions and to generate all necessary binaries and bitstreams for implementation on a modern platform FPGA."

In accordance with the description of work, Task T3.1, deliverable D3.1 provides the following results to the EPiCS project:

> "This deliverable comprises a reference hardware design of the heterogeneous multi-core system specified in WP 1, a basic hard- and software tool flow, and a corresponding report." [DoW WP1 D1.1]

Accordingly, Deliverable D3.1 consists of three parts:

- This report
- Hard- and software tool flow
- Reference hardware designs for the heterogeneous multi-core system

The work on the heterogeneous multi-core architecture builds on our vast experience in reconfigurable hardware, operating system layers, and programming hybrid systems. In particular, the developments made in WP3 and, partly, WP4 leverage our previous architecture and programming environment for hybrid CPU/FPGA systems called ReconOS. In EPiCS we take up and greatly extend basic concepts from earlier ReconOS implementations, provide a new flexible multithreading programming environment for heterogeneous architectures, and target latest platform FPGA technology.

Deliverable D3.1 covers the heterogeneous multi-core architecture and the corresponding basic tool flow. Deliverable D3.2, which covers results of task T3.2 and is due only at month 18, addresses the operating system layer for an autonomous compute node. However, since hardware and software concepts and components in our project are closely connected and interdependent, we incorporate into this report and deliverable D3.1 initial results from Task T3.2. These results encompass the concept of delegate threads representing hardware threads as well as initial low level synchronisation and communication primitives for hardware and software threads.

In the following sections, we first describe the heterogeneous multi-core system (Section 3), focusing on the hardware architecture (Section 3.1), an initial execution model (Section 3.2), and virtual memory support (Section 3.3). Then, we detail our monitoring infrastructure and cores (Section 4). After that, the tool flow for the heterogeneous multi-core is presented (Section 5), comprising ReconOS/eCos and ReconOS/Linux versions. Finally, we describe our reference designs that cover both ReconOS versions (Section 6). The tool flows as well as the reference designs can be accessed through the EPiCS repository.

# 3 Autonomous Heterogeneous Multi-Core Architecture

## 3.1 Hardware architecture

Our heterogeneous multi-core hardware architecture, shown in Figure 1, is built on top of the IBM CoreConnect bus topology available for Xilinx FPGAs. The basic system architecture is independent from the employed host operating system, which is executed in software on the system CPU.

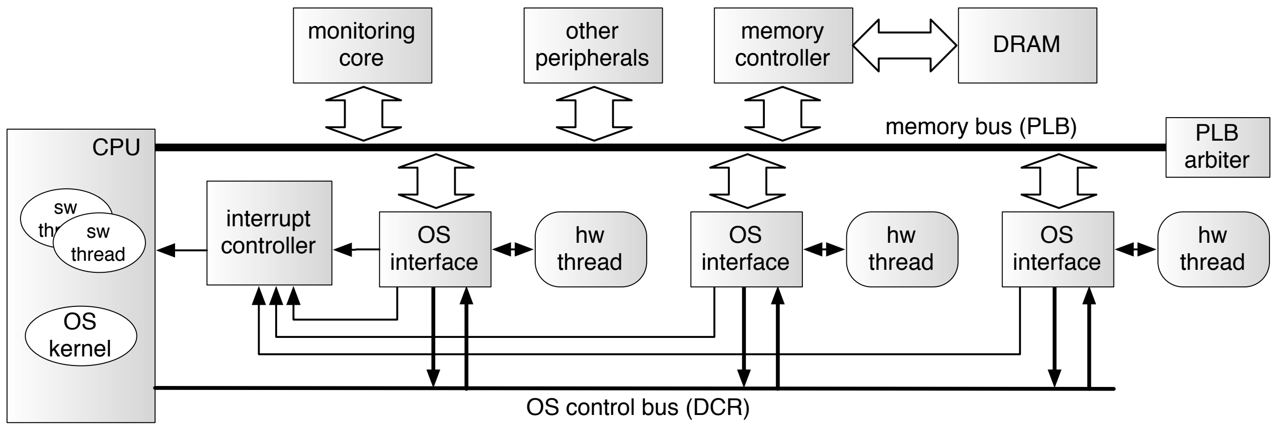

Figure 1: Multi-core hardware architecture with three hardware threads.

Hardware threads are connected to the system via their OS interfaces, which, in turn, are connected to the system's buses. Our heterogeneous multi-cores employ two different buses: the processor local bus (PLB) and the device control register bus (DCR). The 64-bit PLB is used for high-throughput data transfers. Both the CPU cache subsystem and the hardware threads use it to access main memory and system peripherals. All control communication between the OS kernel on the CPU and the threads OS interfaces is routed across a separate 32-bit DCR bus. The separation of control and data communications provides several benefits:

- OS control communications do not obstruct data communications on the memory bus, thus reducing the PLB's arbitration overhead and latency.
- Vice versa, memory communications, especially bursts, can not interfere with OS communications. This reduces the latency of OS calls, which is paramount to the use of our approach in real-time environments.
- OS interfaces for hardware threads that do not need direct access to system memory can be synthesised without the PLB interface, thereby greatly reducing the area footprint. Such threads are typical for many signal processing applications that arrange filter stages in pipeline form, connected by hardware FIFOs.

Based on the hardware architecture shown in Figure 1 and the two OS kernels eCos and Linux, we have created two prototype systems: ReconOS/eCos on a Virtex-6 ML605 FPGA and ReconOS/Linux on a Virtex-II Pro FPGA. The ReconOS/eCos version uses latest platform FPGA technology, the ReconOS/Linux version relies on an older FPGA family. While we have decided to include the stable and well-tested ReconOS/Linux variant on Virtex-II Pro technology with the reference designs of this deliverable, we are also working on a ReconOS/Linux version on Virtex-6. However, due to a major change in the Xilinx soft core architecture, bus systems and Linux kernels, this version requires substantial effort and will be made available at a later point in time.

External SDRAM is used for both the operating system and shared thread memory. The prototypes also include I/O peripherals, such as serial ports, Ethernet interfaces, and general-purpose I/O, all of which are managed by the software operating system kernel.

### 3.1.1 Software threads

The software threads for our multi-core systems as shown in Figure 1 are identical to threads of any host operating system both in concept and implementation. Since software threads are handled by the standard OS scheduler, they are independent from any hardware-related extensions.

Currently, software threads can be implemented using either the eCos kernel API or the POSIX pthreads API–the ReconOS operating system objects can be seamlessly mapped to either API. It is recommended to use POSIX where possible, as it is the more portable API of the two and is also supported via a compatibility layer in eCos. Listing 1 shows an example of a software thread implementing `THREAD_A` using the POSIX API. Here, data is received from a message queue (MQUEUE_IN1), processed, and then copied to shared memory, which is synchronised using semaphores (SEM_READY and SEM_NEW).

Listing 1: Code implementing software thread THREAD A

```
mqd_t mqueue_in1;
sem_t sem_new, sem_ready;
void *shared_mem;

void *thread_a_entry( void *data ) {
    uint8 buf[ MSG_SIZE ];

    while ( true ) {
        mq_receive( mqueue_in1, buf, MSG_SIZE, 0 );
        do_something( buf );
        sem_wait( sem_ready );
        memcpy( shared_mem , buf , MSG_SIZE );
        sem_post( sem_new );
    }
}
```

### 3.1.2 Hardware threads

Software threads have sequential execution semantics. To use an operating system service, a software thread simply calls the corresponding function in the operating system library. Hardware tasks, on the other hand, are inherently parallel. Mostly, there is no single control flow and, thus, no apparent notion of calling an operating system function. In particular, typical hardware description languages, such as VHDL, offer no built-in mechanism to implement *blocking calls*.

To present as unified a programming model as possible to the user, we rely on the following approach: We structure a hardware thread such that all interactions with the operating system are managed by a single sequential state machine. To this end, we have developed an initial operating system function library for VHDL. This library contains code implementing the system call signalling wrapped into VHDL procedures, e.g., `reconos_sem_wait`(). Together with our initial operating system interface (OSIF), a separate synchronising logic module serving as the connection between the hardware thread and the OS, these procedures are able to establish the semantics of blocking calls in VHDL. A hardware thread thus consists of at least two VHDL processes: the synchronisation state machine and the actual user logic. The state transitions in the synchronisation state machine are always dependent on control signals from the OSIF; only after a previous operating system call returns, the next state can be reached. Thus, the communication with the operating system is purely sequential, while the processing of the hardware thread itself can be highly parallel. It is up to the programmer to decompose a hardware thread into a collection of user logic modules and one synchronisation state machine. Besides the increased complexity due to the parallel nature of hardware, this process is no different from programming a software thread.

An example demonstrating this mechanism is illustrated in Figure 2, which again represents THREAD_A from the example above. In this example, the hardware thread receives a message into the local RAM, processes it, waits on a semaphore (SEM_READY), writes the result to shared memory, and then posts another semaphore (SEM_NEW). The OS synchronisation state machine and the user logic communicate via the two handshake signals *run* and *done*.

### 3.1.3 Shared resources

Operating system objects will almost always be shared among different threads. For software threads, this is usually achieved by representing the respective resources as global variables accessible by all threads. This approach creates significant problems when dealing with hardware threads. Since the synthesis tool for a hardware thread written in VHDL cannot easily access the symbol tables of associated software threads, we cannot use global variables defined in software to share an operating system object. Passing a pointer to the data structure representing the OS object to a hardware thread is a possible option; however, it does not replicate the simplicity of the global variable approach.

To provide hardware thread designers with a comparably simple mechanism, we associate an array of resources with every hardware thread. The thread designer can then define integer constants in VHDL which act as indices into the resource array, and use the symbolic constants as arguments to the respective API calls. The definition of the resource array–and thus the mapping between symbolic VHDL constants and actual objects of the operating system kernel–is established at design time. This mechanism also transparently separates the hardware thread API from the API used

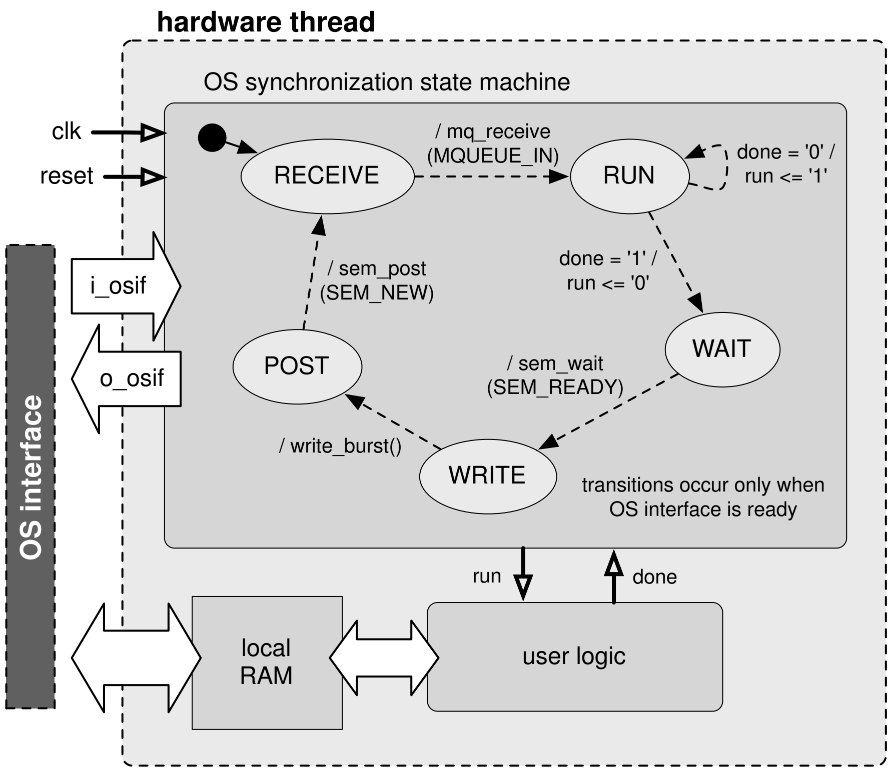

Figure 2: Example of an OS synchronisation state machine

to define the OS objects (e.g., the eCos kernel API or the POSIX API), and it provides a concise overview of the resources used by the individual threads.

Conceptually, hardware threads can directly access the same memory regions as software threads, which allows for efficient sharing of data among threads. As far as the programming model is concerned, all threads share the same address space. To manage concurrent accesses to these memory areas, the threads will usually use synchronisation mechanisms from our multithreading programming model, such as semaphores or mutexes.

In many execution environments, two mechanisms complicate the implementation of shared memory: *caching* and *virtual memory*. Many embedded processors, such as the PowerPC 405 core included in some Xilinx FPGAs, feature a cache unit in the data path between CPU and the memory bus. Sharing an address space between software threads running on this CPU and hardware threads directly connected to the memory bus requires the application designer to explicitly manage cache coherency issues, for example by manually flushing or invalidating cache lines before or after a synchronised data transfer between the threads. This, however, somewhat dilutes the transparency of the programming model. The concept of virtual memory, which is also present in the Linux execution environment (see Section 3.3), further complicates sharing memory. Although software threads usually share the same page tables and virtual memory mappings, this information is not necessarily available for hardware threads. We currently circumvent this problem by allocating a separate, physically contiguous block of memory which is marked as uncachable and advertised to software threads through a memory-mappable file system node. This area is then explicitly used for data exchange between hardware and software threads.

## 3.2 Initial execution model

Hardware circuits modelled as threads and synthesised to an FPGA require a run-time environment to connect them to an existing operating system kernel. On the hardware side, a well-specified interface is required to manage the requests and responses of a hardware thread. On the software side, many of these requests need to be forwarded to the host operating system kernel, and their responses need to be relayed back to the hardware threads. This section details the mechanisms to manage these tasks: the *operating system interface* (OSIF) and the *delegate threads*.

### 3.2.1 Initial operating system interface

To be able to model hardware circuits executing on reconfigurable logic as threads, it is necessary to carefully define mechanisms for low-level synchronisation and communication between the hardware circuitry and the operating system. In our approach, this is the task of the operating system interface (OSIF) which is developed as part of Task T3.2.

### 3.2.2 Delegate threads

A fundamental assumption of our multithreading programming model concerns the transparency of thread-to-thread communication and synchronisation, regardless of the execution context (hardware or software) of the respective communication partners. This enables the designer to easily replace, for example, a software thread with a functionally equivalent hardware thread, allowing for rapid design space exploration with respect to the hardware/software partitioning.

In our approach, every hardware thread is associated with exactly one software thread, its *delegate*, to achieve this transparency. The delegate is responsible for executing operating system calls on behalf of the corresponding hardware thread, making it appear as a software thread to the operating system kernel.

To be able to map the OS objects referenced by the hardware thread to actual instances in the operating system kernel, the delegate thread maintains a table of object instances that are used by the hardware thread (see Section 3.1.3). Individual resources are represented towards the hardware thread as an index into this table. Hence, a single hardware thread description (i.e., VHDL source code, netlist or possibly a relocatable bitstream) can be used for multiple instances in the system; giving different instances access to different resources is simply a matter of changing the delegate's OS object table. This mechanism is also a prerequisite for partial reconfiguration of hardware threads, which is planned as a future extension.

## 3.3 Virtual memory

In general, a system can employ memory virtualisation techniques for different reasons. Giving each process its own virtual address space offers many benefits, such as the simplification of the programming model, and enables features like memory mapping or swapping.

In embedded devices for safety-critical applications, data separation and access control are the predominant reasons for memory virtualisation. While in other domains, these measures may also increase system security (e.g., protect against unauthorised access and ensure data integrity against *malicious* manipulation), in safety-related applications they are indispensable for assuring the safety and availability of critical system functions by guaranteeing spatial separation of functions (e.g., protect against *inadvertent* modification due to errors). For example, certification guidelines for air-borne software systems [1] suggest a functional separation (partitioning) for the purpose of "providing isolation between functionally independent software components to contain and/or isolate faults". Memory virtualisation techniques are also recommended in certification guidelines for integrated modular avionics (IMA, [2]), which aggregate multiple mixed-criticality processes on single processor systems. Consequently, memory address translation is widely supported in existing safety-critical RTOSs, such as PikeOS [3].

Application components with different safety integrity levels are functionally separated into different processes, each with their own OS-managed address translation tables. These tables are stored in protected regions of the main memory and cached by the processors memory management unit (MMU) for low-latency address translation on a per-process basis. This is depicted in Figure 3, which outlines the memory translation mechanisms as implemented in a PowerPC-based embedded Linux operating system. When a process requests a memory access, the virtual address is first split into effective page number (EPN) and offset. The EPN is then translated into the real page number (RPN). In order to map an EPN to a RPN, the operating system maintains process specific page tables. Page table entries are cached in the TLB in order to speed up translation.

Each process may be further split into concurrent threads to parallelise execution (as discussed in the next section); all threads within a process thus share a common address space and, consequently, use the same page translation tables.

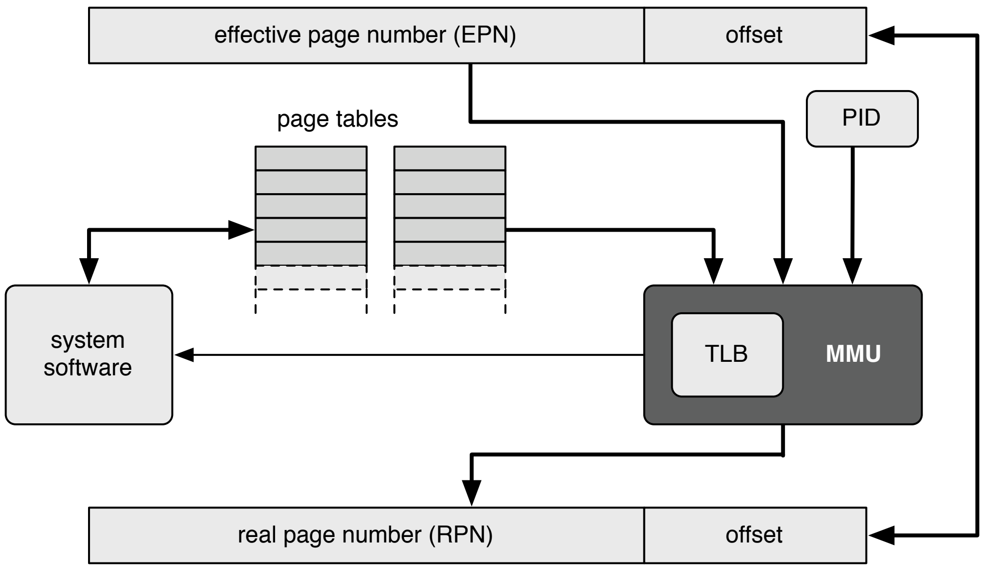

Figure 3: Address translation on a typical embedded system.

### 3.3.1 Virtual memory for hardware threads

In order to enable address space separation and memory access control for hardware threads, we use the concept of a Memory Management Unit, as it is widely used in the domain of general purpose processors. For software-based systems, the MMU is typically closely integrated into the CPU's caching and bus logic. In our approach, hardware threads are connected to the memory bus through the Operating System Interface (OSIF). By integrating the hardware thread MMU (HWT-MMU) into the OSIF, we are able to make use of the already existing communication link between our OSIF and the software operating system so that MMU exceptions can be handled by the delegate threads.

While address translation can be delegated to software, where it can be handled transparently by the operating system, this incurs a large overhead, mainly due to interrupt processing. Instead we chose to supply the HWT-MMU with a state machine that is able to access the operating system's page tables directly. This way, page table entries can be accessed autonomously by each HWT-MMU without interfering with the CPU's processing, which significantly reduces the overhead associated

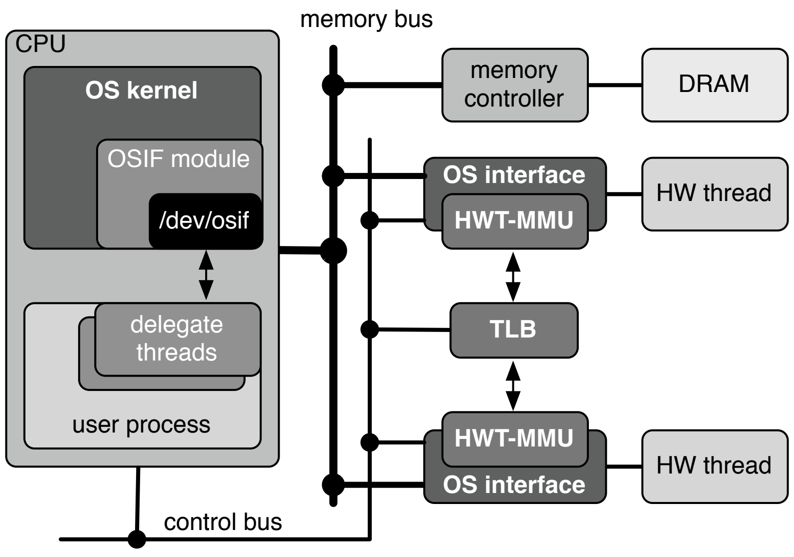

Figure 4: Multiple HWTs connected to a shared TLB.

with TLB misses. Status registers that contain statistics on TLB hits, misses and page faults are attached to the control bus and can be read by software. An overview of the topology of the system is given in Figure 4.

While the HWT-MMU is designed to provide address translation for the HWTs autonomously, exceptions may occur that require further processing in software. There are two possible causes for HWT-MMU exceptions: (1) An access violation exception is generated when the mode of memory access is not permitted, as for instance a write access to read-only memory. Apart from programming errors, a common cause for this is the operating system keeping track of pages that have been written to, by initially marking them read-only. A write access will then trigger an MMU exception and the page can be marked ‘dirty’. (2) A page fault exception may occur when there is no valid page table entry for the page being accessed. While this may be caused by an error in the program resulting in an access to a region in the address space that has not been allocated, page faults may also be generated by the operating system employing demand paging–a strategy that defers the creation of page table entries to the point when the pages are accessed for the first time.

In both cases, the execution of the hardware thread is stopped and the associated delegate thread is notified. The delegate thread will repeat the memory access that triggered the exception. The operating system will then either remove the condition that caused the exception (i.e., marking the page writeable, or creating the page table entry) or–in the case of a programming error–terminate the process. In a safety-critical application the termination of the process gives the system software the opportunity to enter a safe state and take further measures, such as resetting the system or switching control to a backup system. If no error occurs, any cached page tables are flushed to system memory, where they can be accessed by the HWT-MMU and the hardware thread is resumed.

In order to stay consistent with the system page tables, the HWT-TLB is invalidated and the CPU data cache is flushed to system memory whenever the operating system changes or removes existing page table entries. We achieve this by modifying the Linux kernel’s Cache Flush Architecture [4].

# 4 Monitoring Cores

Being able to monitor its internal state is essential for a proprioceptive computing system. Cores running an operating system layer can easily utilise available software functions to retrieve timing and utilisation data. Moreover, most virtual memory enabled operating systems already provide statistics about memory usage and address translation efficiency for software processes; data that can be used to optimise thread scheduling and memory access patterns. To support the monitoring of the internal state also in our heterogeneous multi-core architecture, we have developed two novel monitoring facilities as part of Task T3.1. First, we provide virtual memory monitoring also for hardware threads. Second, we have developed a temperature monitoring core that measures temperature distributions on chip using a thermal sensor grid. The following sections give an overview of these cores. Reference implementations of the monitoring cores can be found in our reference designs which are documented in Section 6.

## 4.1 Virtual memory monitoring for hardware threads

Based on our work on providing virtual memory techniques for hardware threads (for details see Section 3.3.1), we have developed a virtual memory monitoring core that delivers per thread statistics on TLB hits and misses and page faults. Thus we have extend a function that was previously only available for software to the realm of hardware threads. The virtual memory monitoring core has to be built into the hardware design by enabling it in the core configuration panel in the Xilinx Platform Studio. The feature is enabled by setting the property `MMU_ENABLE_STAT_REGS` to `True`. After the design is built, a ReconOS application running on the system can access the virtual memory statistics through the following function call API:

Listing 2: Software API to the virtual memory monitoring core

```
int mmu_get_tlb_hits(reconos_hwthread * hwt);
int mmu_get_tlb_misses(reconos_hwthread * hwt);
int mmu_get_page_faults(reconos_hwthread * hwt);
```

The functions return the number of TLB hits, TLB misses and page faults for the thread represented by `hwt`. The functions are defined in the ReconOS user space library, the declarations are available through the include file `reconos.h`.

## 4.2 Temperature monitoring for multi-cores

Our choice to develop a temperature monitoring infrastructure is driven by the increasing importance for computing systems to perform thermal management. High temperatures influence the switching speed of transistors, which can lead to soft (timing) errors, and contribute to the premature occurrence of hard errors [5]. Over the last years, several dynamic thermal management techniques were discussed, which aim at avoiding hot spots (spots on the chip exceeding certain temperature thresholds) and at balancing the on-chip temperature. One popular approach for thermal management on multi-core systems is thermally-driven thread scheduling and migration which, however, was only shown for homogeneous multi-core systems. We believe that such mapping strategies need to be extended to heterogeneous multi-core systems, e.g., our ReconOS [6], containing processors that execute software threads and reconfigurable hardware slots that execute hardware threads.

We have developed a temperature monitoring core that measures temperature distributions on chip using a sensor grid comprising ring oscillators as thermal sensors. Our heterogeneous multi-core system can autonomously self-calibrate its sensors with the help of internal heat-generating cores. The temperature monitoring core is controlled from ReconOS threads through a software API. In the following, we present the architecture and the software API of our temperature monitoring core. Section 6 presents a reference design for this monitoring core.

### 4.2.1 Thermal sensor grid

We employ a grid of ring oscillators as thermal sensors. A ring oscillator contains an odd number of inverters as depicted in Figure 5 whose output oscillates between '0' '1' at some frequency. The frequency of the ring oscillator changes almost proportional to the temperature, since the switching speed of transistors is directly influenced by the temperature which was shown in [7]. Before a measurement can be done, the oscillator has to 'tune in' for a short amount of time. Then, we count the oscillations within a fixed period of time. The counter value corresponds to the current frequency of the oscillator which can be translated into a temperature value.

In our design, we enable the ring oscillator for $2^{13}$ system clock cycles before measuring. Then, we measure for $2^{17}$ system clock cycles to reduce the impact of the system's noise. Note, that the ring oscillators has to be designed such that its frequency is at most half of the counter's clock frequency in order to provide reliable sampling. Ring oscillators containing 11 inverters have shown good temperature sensitivity when the inverters are mapped to different slices of the FPGA fabric. We map the inverters to defined slices using `LOC` constraints in the user constraint file (UCF) of the FPGA design. In order to get a good sensor coverage of the FPGA's area, we partition the FPGA into a regular grid of tiles and place a sensor at the centre of each tile. Thus, the sensors form a sensor grid. An exemplary sensor grid can be seen in Figure 6(a).

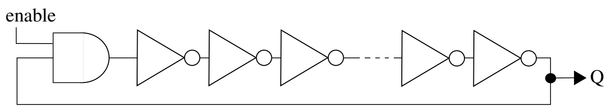

Figure 5: Ring oscillator composed of an odd number of inverters

To be able to correlate measured frequencies to temperature values, the thermal sensors need to be calibrated. Current FPGAs, e.g., Xilinx Virtex-5 and Virtex-6, contain a single pre-calibrated built-in thermal diode. According to [7, 8], for these diodes the relation between measured frequency and temperature is almost linear. However, we are interested in the more involved monitoring of a grid of cores on the chip and, in particular, of spatial gradients. To that end, we have developed a self-calibration approach where we read the temperature diode twice. First, we ensure very low activity on the chip and wait for the temperature to reach a steady-state before reading. Then, we ensure high activity on the chip, wait again for a steady-state in temperature and read the temperature value from the diode again. Taking the two measurements and knowing that the relation between the oscillator frequency and temperature is linear, we compute the frequency-temperature transformation function $T(f)$. It is important to note that we compute an individual frequency-temperature transformation function for each thermal sensor, since the sensors might show different characteristics due to spatial variations of process parameters and supply voltages and also due to different routings. Once the transformation functions are determined, we can measure the temperature distributions on the chip.

To ensure low and high activities on the chip we configure heat-generating circuits, so-called heaters, on the FPGA. Deactivating all heaters results in low activity, while activating them generates high computational load and, thus, high temperature. Figure 6(b) shows an exemplary positioning of such heaters on an FPGA. When activated, a heater toggles 10,000 flip flops at each clock cycle. Using a Virtex-6 FPGA, we are able to generate spatial thermal gradients up to 6.5°C. The local heaters are constrained using area constraints in the user constraint file of the system.

Our temperature monitoring core controls both the heaters and the sensors. The monitoring core is itself connected to a Microblaze CPU via the PLB bus.

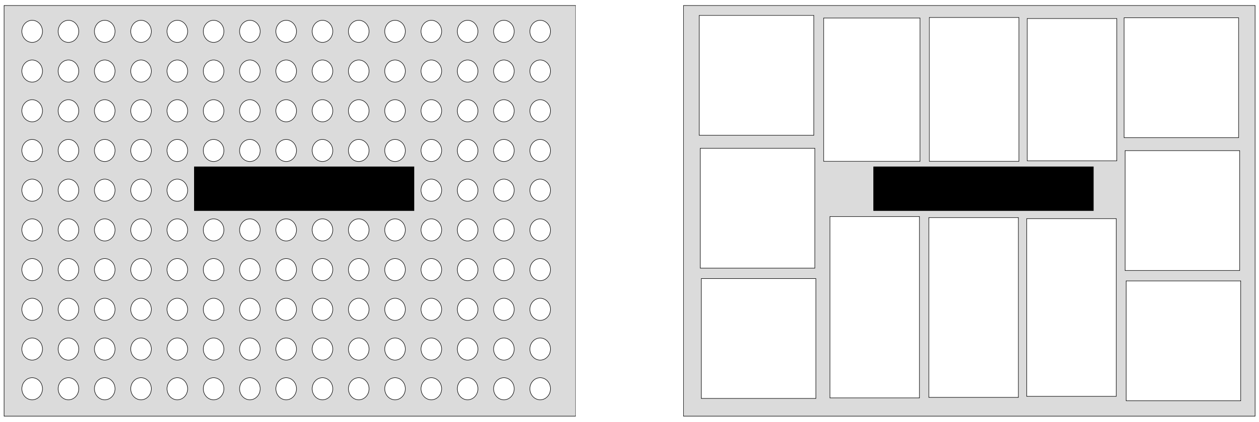
(a) sensor grid (b) heaters

Figure 6: FPGA setup: (a) sensor grid and (b) regional heaters mapped on a Virtex-6 LX240T ML605 FPGA. The black box in the centre is not reconfigurable.

### 4.2.2 Thermal monitoring software API

We provide an application programming interface (API) to access the temperature monitoring core from software, see Listing 3. The API provides a function to self-calibrate the sensor grid using the approach defined above, called `thermal_calibrate_temperature_sensors()`. Furthermore, the API function `thermal_make_temperature_measurement()` triggers a temperature measurement for each sensor in the grid. The temperature of the sensor with the identification number `sensor_id` at the grid location `(x,y)` can be accessed using the functions `thermal_get_temperature(sensor_id)` or `thermal_get_temperature_at_loc(x,y)`. Additionally, the temperature of the internal thermal diode can be accessed by calling `thermal_get_diode_temperature()`. All temperatures are encoded in °C.

Listing 3: Software API to access the temperature monitoring core

```
void thermal_calibrate_temperature_sensors(void);
void thermal_make_temperature_measurement(void);
float thermal_get_temperature(int sensor_id);
float thermal_get_temperature_at_loc(int x, int y);

float thermal_get_diode_temperature(void);

int thermal_get_temperature_sensor_grid_num(void);
int thermal_get_temperature_sensor_grid_width(void);
int thermal_get_temperature_sensor_grid_height(void);

void thermal_activate_local_heater(int heater_id);
void thermal_deactivate_local_heater(int heater_id);
int thermal_get_local_heater_num(void);
```

The function `thermal_get_temperature_sensor_grid_num()` returns the number of temperature sensors in the system, `thermal_get_temperature_sensor_grid_width()` returns the width and, finally, `thermal_get_temperature_sensor_grid_height()` the height of the sensor grid. Each local heater can be de/activated using the functions `thermal_de/activate_local_heater(heater_id)` and the number of local heaters can be read by calling the function `thermal_get_local_heater_num()`.

# 5 Tool Flow

In the following sections we will present an overview of the processes involved in building hardware and software for our system. We then give a summary about the tools which are required to build our heterogeneous multi-core designs.

## 5.1 Tool flow overview

The tool flow for generating the hardware components of a ReconOS system is shown in Figure 7. It starts from a base design prepared for the individual target platform, which is in itself synthesisable and provides a test bed for software-only implementations of the application, as well as a starting point for the iterative design process promoted by ReconOS, in which software threads are subsequently profiled and translated into hardware. The base design is given as a standard Xilinx EDK project, which custom tools modify according to the system specification provided as a project file by the application designer. The applied modifications include instantiations of OSIFs and hardware threads and connections to existing memory and control bus interfaces as well as to the system's interrupt controller. During this process, the tools are capable of adapting to more unconventional system topologies (multiple data buses, etc.). The whole process is source-transparent in the sense that the user can easily modify the system description by hand at intermediate steps of the generation process, for example to allow for custom external thread ports. The resulting completed ReconOS project is then processed by Xilinx' platgen tool, which generates a top-level VHDL description together with VHDL wrappers and a synthesis script for all subcomponents of the system. Using standard FPGA synthesis and implementation tools, we then generate a static bit stream suitable for configuration of the target platform's FPGA.

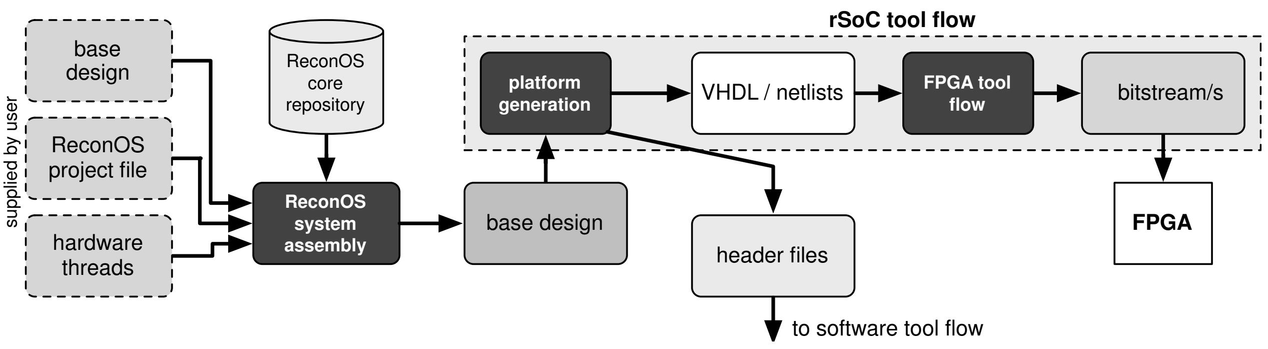

Figure 7: ReconOS hardware tool flow. The platform description needed by the regular reconfigurable System on Chip (rSoC) tool flow is automatically generated from a ReconOS application description.

The ReconOS repository provides a number of base designs for our prototype target platforms. Depending on the requirements of the application, these base designs contain various combinations of peripherals, such as network, display, and other I/O controllers, and consequently offer different amounts of free logic resources to be used for the implementation of hardware threads. From the hardware perspective, supporting a new target platform is simply a matter of adding a new base design, which can usually be derived directly from manufacturer-provided example designs or board support packages (BSPs). To be suitable for the generation of a static ReconOS system, a base design must at least include a CPU supported by the host operating system, a main memory bus, a control bus, and an interrupt controller. Different communication topologies are possible, but may need manual system assembly or a suitable porting of the system generation tools.

Hardware threads are described by one or more VHDL files, with the hardware thread's top level entity including the OSIF interface signals. The thread description is then wrapped in a Xilinx EDK compatible IP core description, which is later incorporated in the system assembly process. The generation of the EDK wrapper is automatically handled by the tool chain, which supports hardware threads written purely in VHDL as well as thread descriptions containing both VHDL code and pre-synthesised netlists.

Depending on the way a target software application and its host operating system is assembled, ReconOS provides its software API and functionality either through OS integrated packages or through kernel drivers and a user-space library. The following paragraphs describe the integration of the respective software components into the host operating systems eCos and Linux.

### 5.1.1 ReconOS/eCos

Due to the fact that eCos is targeted at the embedded software segment, applications deployed using eCos are represented by a single, monolithic executable, rather than a kernel, dynamic libraries, and run-time loadable process images. In other words, an eCos system typically runs a single process at a time, and generally does not allow dynamic loading of code that was not present at system compile-time. Consequently, the compilation of an eCos application involves first creating a static library containing all code relating to operating system services, which is then linked against the compiled object files of the actual application using these services.

The major differentiating feature of eCos is its configurability at the source code level. Using an elaborate system of hierarchic configuration files (specified in a language called CDL) and several custom tools, application developers can specify the feature set of the created eCos library at a fine granularity and without incurring run-time overheads.

eCos groups related functionalities, such as POSIX compatibility functions, maths libraries, networking protocol stacks, or hardware drivers in logic groups called packages. In order to provide ReconOS functionality to eCos developers in a transparent way, we have started to integrate its functionality–consisting of thread creation API, data structures, initialisation functions, delegate thread code, and hardware interface routines–in its own eCos package. The package can be configured in the same fine-granular way as other eCos packages to exclude selected ReconOS functionality not required by an individual application.

During compilation, a parsing tool integrated via CDL files into the eCos build process generates the delegate thread code from the command definition files. Also, hardware parameters such as OSIF memory addresses and interrupt vectors are extracted from the modified hardware design generated in the hardware implementation step. After compiling and linking both the eCos library and the application code, the resulting executable is typically loaded using the FPGA's JTAG debugging interface, although other mechanisms, such as boot loaders, are also feasible. Figure 8 depicts the complete software tool flow for a ReconOS/eCos system.

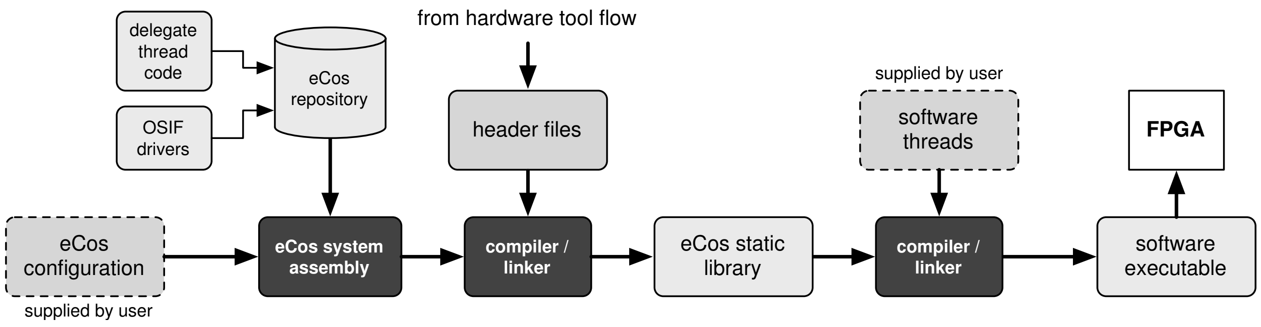

Figure 8: ReconOS/eCos software tool flow. The eCos software repository is extended with delegate routines and OSIF drivers and statically linked to the user's software source code.

### 5.1.2 ReconOS/Linux

In contrast to eCos, a Linux system provides a vastly more flexible, and consequently more complex, execution environment for software. Instead of statically linking application code and OS functionality during compile-time, most Linux applications are contained in independent executable process images, which interact with operating system services through special system calls provided by dynamic libraries and the Linux kernel run-time environment. To execute a Linux application on an embedded system such as our prototype FPGA boards, we therefore need to load and boot the kernel first before logging into the system and executing the application. Thus, the compilation of the kernel and of the ReconOS application are two independent steps, depicted in Figure 9.

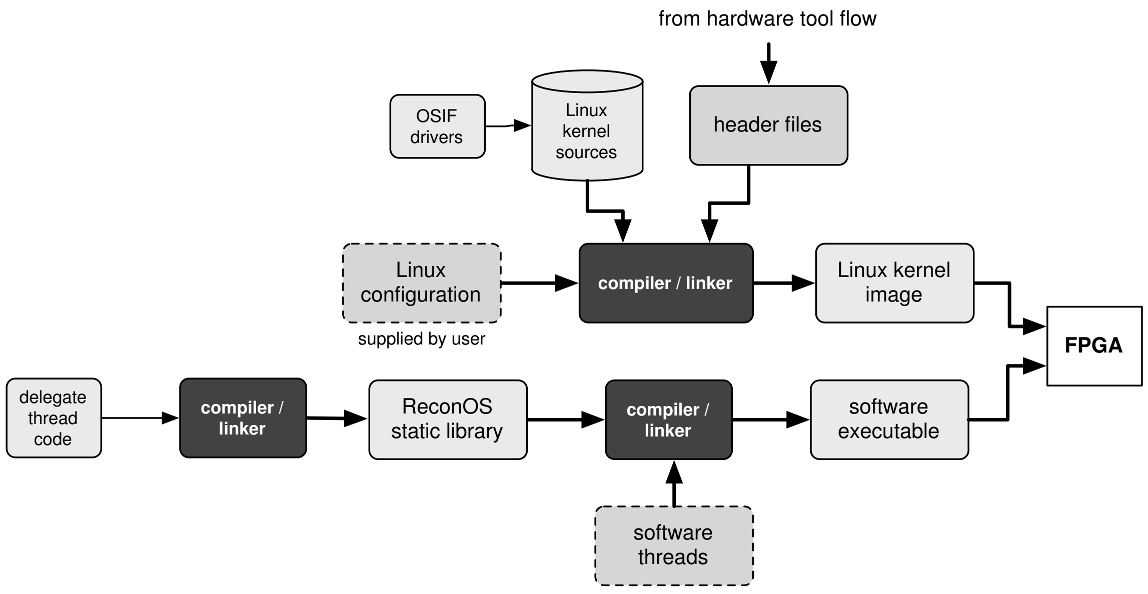

Figure 9: ReconOS/Linux software tool flow. The OSIF driver code is integrated into the kernel. The delegate thread code runs in user space and is linked as a library to the user application.

The ReconOS-specific functionality is split between a kernel module and a user-space library. The kernel module, which provides delegate threads with a means to access OSIF hardware registers can either be compiled statically into the kernel, or loaded dynamically after booting the system. Unlike the eCos OSIF interface code, the kernel module retrieves hardware parameters, such as memory addresses and interrupt vectors, at boot or load time from the Linux device tree. This allows changing the hardware memory layout without recompiling the Linux kernel, significantly shortening the software development cycle.

The ReconOS user-space library contains the delegate thread's code, which interacts with the hardware through the kernel module, as well as the thread management API and related data structures. Because the library provides generic interfaces both toward the software (via the predefined API) and the hardware (via the kernel module), it does not need to be recompiled when changing the hardware design or the software application. The actual ReconOS software application code uses the defined API provided by the user-space library, which is identical to the API provided by the eCos package from the previous section, and thus is exactly the same for both Linux and eCos targets.

## 5.2 Prerequisites and installation

The ReconOS tool chain is developed to run on Unix/Linux systems and depends on a few other tools, both commercial and open-source. While we try to rely on as much free software as possible, some of the tools (most notably the vendor specific FPGA synthesis and implementation tools) are not available for free.

The following commercial vendor tools for FPGA synthesis, implementation, and simulation must be installed:

- Xilinx ISE and EDK version 12.3. If you use a Xilinx Virtex-2 FPGA you must install a previous version such as Xilinx ISE and EDK version 10.1.
- ModelSim SE version 6.1f or higher. This tool is actually not necessary, but greatly recommended for hardware thread and system simulation as well as automated regression testing.

ReconOS heavily relies on Python for the automated parts of its tool chain. The following scripting languages and parsers are required (Our production system uses the specified versions; newer versions of the software may work as well):

- Python version 2.5.1
- pyparsing 1.5.1
- Cheetah 2.0.1
- Cross-compilers, binutils and libraries

For building ReconOS/eCos systems you have to install eCos version 2.0.

To install a ReconOS development environment, clone the development git repository `http://github.com/EPiCS/reconos`:

```
git clone https://github.com/EPiCS/reconos
```

To set all the environment variables and to be able to use the ReconOS scripts execute the `setenv.sh` script in the repository's root directory:

```
source setenv.sh
```

# 6 Reference Designs

In order to provide a quick start for anyone who wants to work with ReconOS we provide two reference designs as entry points. The reference designs–one for ReconOS/eCos and one for ReconOS/Linux–contain fully working Systems on Chip, featuring serial IO, networking, SDRAM support, and embedded PowerPC CPUs as well as Microblaze soft core CPUs. The designs also contain exemplary hardware threads and the monitoring cores described in Section 4. Table 1 gives an overview of important reference design features.

Table 1: Reference Design Feature Overview

| Feature | ReconOS/eCos | ReconOS/Linux |
|---|---|---|
| Platform | Xilinx ML605 | Xilinx Virtex2Pro XUP |
| FPGA | XC6VLX240T | XC2VP30 |
| System CPU | Microblaze v7.30b | PowerPC 405 |
| CPU Clock | 100 MHz | 300 MHz |
| MMU | No | Yes |
| Host OS | eCos | Linux |
| IO | Serial | Serial<br>Ethernet |
| Monitoring Cores | Temperature | Virtual Memory |
| Application | Sorting | Performance Benchmark |
| Number of HW-Threads | 1 | 2 |

## 6.1 ReconOS/eCos

The ReconOS/eCos reference design provides a fully operational SoC design for the Xilinx Virtex-6 LX240T FPGA ML605 Evaluation Kit. The reference system contains a Microblaze soft core processor that handles the operating system eCos including the ReconOS package, a ReconOS hardware thread, and a temperature monitoring core. The system contains a sort demo application. In a first step the application generates random data that is divided into chunks of 8 Kbyte in a second step. These chunks are sorted by the hardware thread using the bubble sort algorithm. The chunks are then merged such that in the end the entire data is sorted. To ensure correctness, the application checks whether the data has been sorted correctly. These steps run in an iteration.

The ReconOS/eCos reference design also makes use of our temperature monitoring infrastructure. We have partitioned the employed FPGA (LX240T) that comprises 160x239 logic slices into a regular grid of 10x15 tiles where each grid contains a temperature sensor at its centre. On this FPGA there is a central region which is not reconfigurable and, thus, cannot be used for temperature sensor placement. In total our reference design contains 144 temperature sensors, as shown in Figure 6(a). Furthermore, we have implemented 12 heat-generating circuits and constrained them to disjunct areas that can be seen in Figure 6(b). In between different iterations of the sorting application, the temperature distribution of the FPGA chip is written to the serial output port.

The reference design consists of the following components:

- Reference hardware design without ReconOS pcores and hardware threads located at `support/refdesigns/12.3/ml605/ml605_light_thermal`
- ReconOS specific pcores located at `core/pcores`
- Application including ReconOS software and hardware threads `demos/sort_demo_thermal`

- ReconOS/eCos package for eCos v. 2.0 located at `core/ecos/ecos-patched/ecos/package/`
- eCos version 2.0 can be installed from `ftp://ecos.sourceware.org/pub/ecos/releases/ecos-2.0/ecos-2.0.i386linux.tar.bz2`

All paths are relative to the root of the ReconOS repository snapshot that can be found in the EPiCS repository as part of the Deliverable D3.1. A user guide which provides information how to build the reference design can be found in the reference design directory (`support/refdesigns/12.3/ml605/ml605_light_thermal/user_guide_ecos.pdf`).

## 6.2 ReconOS/Linux

The ReconOS/Linux reference design provides a fully operational SoC design for the Xilinx XUP2 board. It uses the PowerPC 405 CPU embedded in the XC2VP30 FPGA as the main CPU. Additionally, we provide a modified operating system kernel as well as a user space library against which ReconOS/Linux applications can be build. The system already contains a demo application that uses two hardware threads and one software thread. The application is meant to demonstrate the use of a TLB that is shared among two hardware threads.

The reference design consists of the following components:

- Reference hardware design located at `demos/shared_tlb_demo/opb_eth_cf`
- Demo application hardware thread sources at `demos/shared_tlb_demo/hwt`
- Demo application software thread sources at `demos/shared_tlb_demo/shared_tlb_demo.c`

All paths are relative to the root of the ReconOS repository snapshot that can be found in the EPiCS repository as part of the Deliverable D3.1. A user guide which provides information how to build the reference design can be found in the reference design directory (`demos/shared_tlb_demo/user_guide_linux.pdf`).

In addition to the reference design sources the following components of the ReconOS runtime system are needed to build the design:

- MMU enabled OSIF core including monitoring extensions located at `core/pcores/osif_core_v2_03_a`
- TLB core located at `core/pcores/osif_tlb_v2_01_a`
- TLB arbiter core allowing multiple hardware threads to share a single TLB: `core/pcores/tlb_arbiter_v2_01_a`
- ReconOS/Linux kernel patch located at `core/linux/kernel/linux-2.6.27-rc9-xlnx.patch`
- ReconOS/Linux kernel module located at `core/linux/drivers/reconos`
- ReconOS/Linux user space library located at `core/linux/libreconos`

## References

[1] RTCA, "Software Considerations in Airborne Systems and Equipment Certification (DO-178B)," 1992, http://www.rtca.org.

[2] ——, "Integrated Modular Avionics (IMA) Design Guidance and Certification Considerations (DO-297)," 2007, http://www.rtca.org.

[3] R. Kaiser and S. Wagner, "Evolution of the PikeOS Microkernel," in *1st International Workshop on Microkernels for Embedded Systems*, 2007.

[4] D. S. Miller, "Cache and TLB Flushing Under Linux," Linux Kernel Documentation, December 2008, linux/Documentation/cachetlb.txt.

[5] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE MICRO*, pp. 10–16, November/December 2005.

[6] E. Lübbers and M. Platzner, "ReconOS: Multithreaded Programming for Reconfigurable Computers," *ACM Transactions in Embedded Computing Systemy (TECS)*, vol. 9, 2009.

[7] S. Lopez-Buedo, J. Garrido, and E. I. Boemo, "Dynamically Inserting, Operating, and Eliminating Thermal Sensors of FPGA-Based Systems," *IEEE Transactions on Components on Packaging Technologies*, vol. 25, no. 4, pp. 561–566, 2002.

[8] S. Velusamy, W. Huang, J. Lach, M. Stan, and K. Skadron, "Monitoring Temperature in FPGA based SoCs," in *Proceedings of the IEEE Int. Conf. on Computer Design*, 2005.