Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2011,1
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

The OpenProcessor Platform

Fostering Research on the Hardware/Software Boundary

Raphael Neider

2011

The OpenProcessor Platform

Fostering Research on the Hardware/Software Boundary

Raphael Neider
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5
76128 Karlsruhe, Germany
neider@kit.edu

ABSTRACT

Today's computing systems typically execute task-specific software on general purpose hardware. As a consequence, the software layer must make do with whatever instructions, information, and services the underlying hardware exposes. This restriction can lead to implementing inefficient approximations rather than the intended functionality of software. For example, virtual memory subsystems in operating systems often approximate the desired *least recently used* (or *least frequently used*) strategy for page replacement using a two-handed clock algorithm just because the hardware only provides a single referenced bit rather than the required timestamp (or reference counter) per page.

Programmable hardware such as sufficiently large field programmable gate arrays (FPGAs) has been available for quite some years and could be used to give software developers the possibility to alter or augment the hardware to better match their demands. However, there is no basic system available that can serve as a starting point for FPGA-based HW/SW research and that is both sufficiently powerful and easy enough to work with.

In this report, we describe the OpenProcessor platform, a free and open source computing platform, comprising both synthesizable hardware and support software. The hardware is described in Verilog, ready for deployment on an FPGA development board, and designed with a focus on extensibility and simplicity rather than utmost performance in order to attract many developers. The software stack comprises a microkernel-based operating system offering a subset of the POSIX API to execute on the FPGA plus a number of management and monitoring services to run on an attached PC.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: hardware/software interfaces, instruction set design (RISC), modeling of computer architecture; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, tracing*; D.4.9 [**Operating Systems**]: Systems Programs and Utilities; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*

General Terms

Design, Experimentation, Measurement

Technical Report 2011,1
Department of Informatics
Karlsruhe Institute of Technology (KIT)

1. INTRODUCTION

Today's computers comprise both hardware and software. Of these two, software – even system software such as the Linux or BSD kernels – is relatively easily accessible for a large group of people, as the required tools (a computer with a text editor and a compiler) are inexpensive and the required software development skills are provided by schools, high schools, and universities. As a consequence, many developers can and do continuously devise, implement, and evaluate improvements to open source software and contribute their findings to the community of developers and users, causing rapid progress of software features.

Conversely, hardware development traditionally required access to expensive development tools (both software and manufacturing facilities) and was thus performed by only very few developers – typically employees of hardware manufacturers such as Intel or AMD or hardware design companies such as ARM. Apart from the lack of tools, hardware development skills were also not as common as software development skills are – probably due to the fact that they were of little use for the common developer who did not have the means to actually implement and evaluate her potential hardware improvements. Thirdly, we are unaware of any complete open source hardware platform that could serve as a starting point for interested hobbyist developers.

As a consequence of this imbalance in development force on the software and the hardware sides, software has successfully exploited most if not all of the hardware features available and is now somewhat limited in its progress by the lack of hardware support. As an example, many operating systems approximate the theoretically optimal *least recently used* (or *least frequently used*) page replacement policy using a two-handed clock algorithm, which rather implements a *recently not used* policy. The need to rely on this approximation stems from the hardware, which commonly only provides a single "referenced" bit per page instead of true timestamps (or access counters or access frequencies). If hardware development could and was to be done by more people, the chance of finding efficient and effective hardware improvements to better support software would greatly increase.

Over the past years we have seen an ever increasing availability of sufficiently large reconfigurable hardware devices such as FPGAs, which allow developers to actually implement and evaluate non-trivial hardware systems – even at home. At the same time, high schools and universities have begun teaching hardware development skills, e.g., by offering courses in the Verilog or VHDL hardware description languages and even courses in FPGA-based hardware design. As a consequence, hardware development could now be carried out (or at least be supported) by a larger community not unlike the software development communities. As a matter of fact, an active open source hardware community already exists, as evidenced by websites such as OpenCores.org [2].

However, one of the major problem remains: Most commodity hardware is still "closed" and kept a trade secret of its developers and is thus not open for modification and experimentation by such a community. Even if companies such as Intel or AMD decided to publish their hardware designs, these would probably be too complex for the casual developer to be able to contribute.

Thus, we see demand for a rudimentary, modular computing system that is at the same time

1. powerful enough to be of use,
2. simple enough to attract a large number of developers, and
3. small enough to fit into today's reconfigurable hardware devices to facilitate testing and evaluating modifications.

Our contribution is the OpenProcessor platform, a completely synthesizable computing environment including a processor core and device controllers for implementation on an FPGA development board. Additionally, the platform provides supporting software tools such as a compiler toolchain based on GNU GCC [1], a bootloader, and a microkernel-based operating system that offers a subset of the POSIX API to run applications on. All components are completely open source and simple enough to facilitate experimentation by anyone interested, which has been successfully demonstrated by a number of students who have developed and implemented modifications on core features of the platform each in three to six months.

The rest of this report is organized as follows: In Section 2, we provide an overview of the hardware of the OpenProcessor platform with a focus on the design decisions that were made to keep even low level aspects such as virtual memory support of the platform easily exchangeable and/or customizable. Following the basic platform overview, we will point out some of the major monitoring and debugging means available on the OpenProcessor platform to facilitate performance evaluation down to the level of single instructions in Section 3. In Section 4, we briefly describe the microkernel-based operating system *KaOS* that we have co-developed with the hardware to explore the requirements of modular operating systems and that can serve as a starting point for experiments in the realm of system software or hardware/software co-design. Related work with respect to reconfigurable hardware and microkernel-based systems is presented in Section 6, before we conclude in Section 7 with a summary and a glance at future work.

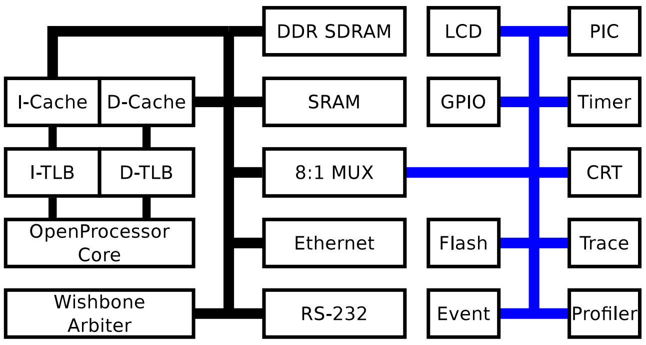

Figure 1: The OpenProcessor platform comprises a processor and several device controllers, connected via a hierarchical Wishbone bus.

2. OPENPROCESSOR PLATFORM

The OpenProcessor platform provides a rudimentary yet fully functional computing system, implemented in the hardware description language Verilog and targeted at being instantiated in a field programmable gate array (FPGA).

As a physical basis for the system we use an off-the-shelf Memec 4VLX60MB development board, which features a sufficiently large Virtex 4 LX60 FPGA at its core. The FPGA is connected to 64 MiB of DDR SDRAM, 4 MiB of Flash memory, an RS-232 connector, an RS-232–USB-bridge, a 100 Mbit Ethernet connector, and a 2x16 character LCD as well as to four LEDs, four pushbuttons and eight DIP switches for user interaction. The board also includes a clock generator, which is used to supply the OpenProcessor system in the FPGA fabric with a 50 MHz clock.

The OpenProcessor platform (see Figure 1) provides implementations of controller modules for each of the above mentioned devices except the clock generator, which is configured via a separate set of DIP switches. The device controller modules are connected to the OpenProcessor core using the Wishbone interconnect [13] in its bus configuration, though most controllers are connected to the Wishbone bus via an adapter module to facilitate a change of the interconnect in the future (e.g., to compare bus, ring, and point-to-point interconnects while keeping the rest of the system unchanged). The platform also provides support for virtual memory via software-managed translation look-aside buffers with address space tags (tagged TLBs) as well as write-back caches for cacheable memory pages.

In the following sections, we will present the design of the CPU core, followed by a discussion of the virtual memory management scheme and the fundamental I/O devices in the system.

2.1 OpenProcessor Core

At the heart of the OpenProcessor platform lies the OpenProcessor core, a pipelined, 32 bit MIPS-like RISC processor that has been designed to be easily understandable without sacrificing too much performance.

2.1.1 Register Set

The OpenProcessor core offers 64 32 bit registers, split into 32 general purpose registers (R0–R31) for use by the software and 32 special purpose registers (R32–R63) to facilitate interrupt and exception handling as well as privilege management (see Section 2.1.5), and to provide basic execution statistics (see Section 3.1). The default register usage is shown in Table 1. In addition to these externally visible registers, the CPU has an internal STATUS register to hold the traditional arithmetic status flags (parity, sign, zero and carry), and an internal instruction pointer register, which is incremented after each instruction unless it is explicitly redefined by a taken branch instruction (see below).

Register	Purpose / default usage
R0	always reads as 0
R1	reserved for kernel-mode entry
R2–R15	callee-saved registers
R2	stack pointer
R3	frame pointer (if desired)
R16–R31	caller-saved registers
R27	fourth argument
R28	third argument
R29	second argument
R30	first argument
R31	return address (set by CALL instructions)
R32–R47	privileged registers
R32[‡]	saved STATUS register
R33	saved instruction pointer
R34[‡]	saved next instruction pointer
R35	faulting address (if memory related)
R36	fault reason
R37	fault information
R38	opcode of the faulting instruction
R44[†]	kernel ASID
R45[‡]	user ASID
R46[†]	kernel config
R47[‡]	user config
R48–R63	read-only registers (for benchmarking)
R48	hold register (upper 32 bit of R56–R63)
R56	number of operand fetch stall cycles
R57	number of TLB-related stall cycles
R58	number of D-cache related stall cycles
R59	number of I-cache related stall cycles
R60	number of taken branches
R61	total number of instructions executed
R62	total number of stalled cycles
R63	total number of clock cycles since start-up

Table 1: The OpenProcessor platform implements 32 general purpose and up to 32 special purpose registers.
[†]: read when (re-)entering kernel mode,
[‡]: read when returning to user mode

2.1.2 Instruction Encoding

As shown in Table 2, instructions are encoded in 32 bit words with only five similarly structured formats. The formats are laid out to make instruction decoding easy: The instruction class, which mostly identifies the structure of the opcode, is always encoded in the three most significant bits of the instruction, followed by a class-dependent number of bits to identify the operation. Indices of the register operands (if used) are always located in bits 16–21, 10–15, and 0–5, and immediate operands (if used) are always located in bits 0–9 or 0–15, again depending on the class of the instruction.

Five of the eight instruction classes are used to implement the basic integer-only instruction set, the remaining three classes can be used to implement extensions such as the EVENT instruction (see Section 3.2).

Instructions in classes 0–2 provide the common set of integer arithmetic operations (ADD, ADD with carry, SUB, SUB with borrow, and a 32 bit $\times$ 32 bit $\rightarrow$ 32 bit MUL, but *no* DIV due to its complexity in hardware), bitwise and logic operations (AND, OR, XOR, NOT, plus bit set/reset/toggle instructions), sign- and zero-extensions of 8 bit or 16 bit values, and arithmetic as well as logical shift and rotate operations (SAL, SAR, SLL, SLR, ROL, ROR). Class 0 instructions address two source registers and a (third) destination register. Instructions in class 1 address one source register and a destination register and take their second operand from a 10 bit (signed) immediate embedded in the opcode. Class 2 addresses only one register, which serves as both the first operand and as the result register, and carries a 16 bit immediate to be used as the second operand.

All of these instructions also update the STATUS register according to the result of the operation.

Memory access is implemented in class 3: Aside from sign-extending LOADSN and zero-extending LOADUN instructions to read $N \in \{8, 16, 32\}$ bit entities, this class also covers unconditional STOREN as well as *guarded* STOREGN instructions to conditionally write $N \in \{8, 16, 32\}$ bit from the N least significant bits of the first register operand to memory. Inspired by MIPS' linked load/store conditional instructions [15], the guarded stores only succeed if no other thread (or CPU/DMA device) modified *any* memory location between a previous LOAD and the guarded STORE. A sequence of LOAD and guarded STORE instructions can thus be used to implement atomic read-modify-write memory cycles, which in turn can be used to implement synchronization facilities such as (counting) semaphores. This simple, address-agnostic scheme works well with only one CPU in the system, as guarded STOREs will only fail if an interrupt or exception has occurred between the LOAD and the STORE. When multiple CPUs access a shared memory, the address of the LOAD and the guarded STORE should probably match and be watched for external modifications, so that guarded STOREs fail less often.

Memory accesses must be naturally aligned; otherwise the result is undefined. For simplicity, the CPU does not raise an exception for unaligned memory accesses, but rather passes on whatever result is delivered from the memory implementation. In practice, the compiler will make sure that all

Example	Semantics	Encoding scheme (MSb on the left)
ADD RZ, RA, RB	RZ ← RA + RB	`ccc.oooooooo.aaaaaa.bbbbbb.----zzzzzz`
ADDJ RA, RB, J	RA ← RB + J	`ccc.oooooooo.aaaaaa.bbbbbb.jjjjjjjjjj`
ADDI RA, J	RA ← RA + J	`ccc.oooooooo.aaaaaa.jjjjjj jjjjjjjjjj`
LOADU16 RA, (RB), J	RA ← MEM[RB + J] AND 0xFFFF	`ccc.oooooooo.aaaaaa.bbbbbb.jjjjjjjjjj`
JCCI M, V, J	skip J instructions iff STATUS AND M == V	`ccc.ooo.mmmmm.vvvvv.jjjjjj jjjjjjjjjj`
CALL M, V, (RB), J	conditionally call the function at RB+4J	`ccc.ooo.mmmmm.vvvvv.bbbbbb.jjjjjjjjjj`

Table 2: The OpenProcessor instructions are encoded using one of only five formats to simplify decoding. c: class, o: operation, a/b/z: first/second/third register operand, j: (sign-extended) immediate, m/v: mask and required value of status bits for conditional branches, –: don't care

objects are suitably aligned unless they are under the influence of the `packed` attribute, in which case GCC will ignore alignment requirements and access the objects byte-by-byte. The latter case results in fairly costly, but correct memory accesses. See our discussion of the Ethernet module in Section 2.3.4 for an example of how I/O hardware can help to meet alignment requirements.

Class 7 provides conditional branches and calls, each of which can either be relative to the current instruction pointer (using a 16 bit signed displacement) or specify the absolute destination address in a register. Each branch is predicated with a 5 bit mask to be applied to the STATUS register and a 5 bit vector of values and is taken only if all selected bits equal the given values. To set up the STATUS register, conditional branches are typically preceded by a SUB instruction to effectively compare two values.

All branch instructions have a single branch delay slot [20], meaning that the instruction that follows the branch is executed regardless of whether the branch is actually taken or not. Such branch delay slots are common on in-order pipelined architectures and serve to hide the latency between loading the branch instruction and recognizing that it actually is a taken branch. Without branch delay slots, all instructions loaded in the meantime would have to be discarded; declaring (a subset of) them as branch delay slots thus improves runtime efficiency.

Notable design decisions regarding the instruction set include the omission of any of the common special purpose instructions for TLB- and/or cache management such as MIPS' TLBRD or TLBWR instructions or Intel's WBINV. Both TLBs and caches are regarded as (more or less) regular I/O devices using the memory mapped I/O approach, which can accommodate more extreme changes to the interface of the TLB or cache modules and simplifies CPU design by further decoupling the CPU from the memory hierarchy.

2.1.3 Execution Pipeline

As illustrated in Figure 2, the OpenProcessor core features an eight-stage pipeline including result and load forwarding. The pipeline is derived from the classic five-stage RISC pipeline [19] comprising instruction fetch (IF), instruction decode and operand fetch (ID/OF), execution (EX), memory access (MA), and write back (WB) stages, but differs notably in several aspects.

The classic five-stage pipeline is based on the assumption that memory accesses can succeed in a single cycle, so that one MA cycle suffices. However, our current cache implementation requires two cycles for memory writes even on cache hits: The first cycle is used to determine a hit on the cache, and only after the hit has been ascertained will the second cycle be used to update the data in the cache. In order to allow for proper pipelining of all memory accesses, both reads and writes take two cycles in the cache module. Though a specialized read-only implementation for the instruction cache might provide single-cycle access to instructions in the future, we currently use the same cache implementation for both the instruction and the data path, thus increasing the pipeline length by two stages, I-Cache and D-Cache, to seven stages.

Another decision that has been made in favor of simplicity over performance is the lack of a first level (L1) cache that could be accessed without requiring prior virtual-to-physical address translation via the TLB: Such "virtually indexed, virtually tagged" caches suffer from aliasing effects [7], i.e., multiple virtual addresses can refer to the same physical memory location, which can lead to multiple copies of a single physical datum residing in the cache at the same time. In such a scenario, the L1 cache logic must avoid data inconsis-

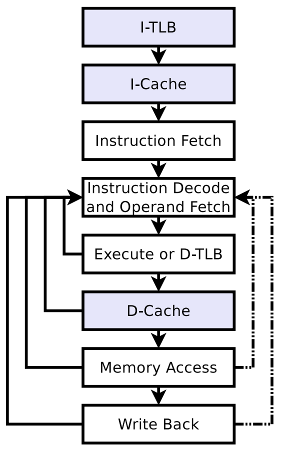

Figure 2: The OpenProcessor pipeline comprises eight stages: three for "instruction fetch", one for both "instruction decode" and "operand fetch", three for "execute" and "memory access", and one for "write back". The arrows on the {left|right} show available {result|load} forwarding paths.

tencies either by ensuring that updates to one of the copies invalidate the other copies, or by updating all copies of the data in the cache simultaneously, or by avoiding duplicate data in the cache in the first place.

In its present state, the OpenProcessor platform avoids the aliasing problems by not incorporating any L1 caches; it only provides "physically indexed, physically tagged" second level (L2) caches that do not suffer from aliasing effects as they operate on physical addresses. On the downside, access to L2 caches requires the incoming address to be translated from the virtual to the physical domain before use, which requires at least one additional cycle per memory access and thus reduces performance. To counter this effect, the pipeline of the OpenProcessor core includes one I-TLB stage and one D-TLB stage to perform the virtual address translation just before the instruction and data caches are accessed. As a result, the core retains single-cycle throughput of all instructions including LOAD and STORE in the common case of hits both in the instruction and (if required) in the data cache, but pays for this property with an increased instruction latency of two more cycles, I-TLB and D-TLB, raising the pipeline length to nine cycles.

Due to the inclusion of all required address computation logic into the ID/OF stage, memory addresses need not be computed in the EX stage, so that the D-TLB stage can be overlaid with the EX stage to decrease the pipeline length to the announced eight cycles. The effectively three additional stages as compared to the classic pipeline are shaded blue in Figure 2.

2.1.4 Forwarding and Stalling Logic

Many pipelined architectures – including the OpenProcessor core – apply load and result forwarding [8] (or *bypassing*) to hide latencies between loading a word from memory or computing an arithmetic result and making it available through the register file.

In the classic five-stage RISC pipeline, the result of the arithmetic and logic unit (ALU) is available by the end of the EX stage, but stored into the register file only two cycles later by the end of the WB stage. Without forwarding logic, the updated values will thus be visible to following instructions only once the providing instruction has passed its WB stage. If the immediately following instruction already requires the updated register as an input, this instruction will either read the old value or must be delayed (or *stalled*, using a *pipeline interlock* [19]) until the updated result is available from the register file. Result forwarding logic cuts this time short by providing ALU results from instructions in or before their WB stage as an alternative input to the ID/OF stage.

As shown in Figure 2, the forwarding logic of the OpenProcessor needs to deliver ALU results from the (new) D-Cache stage in addition to the EX, MA, and probably WB stages, whence forwarding would be required for the classic pipeline as well. This does not cost performance, but makes the forwarding logic more complex in terms of hardware resources. However, as implementing this logic yields fairly regular code along the lines of "if stage X holds a new value for register Y, forward it to ID/OF", the perceived complexity for the hardware developer does not increase much with an increasing number of stages, so that we rather accept complicating the forwarding logic than implementing single-cycle L1 caches.

Similar to result forwarding, load forwarding logic serves to provide the results of LOAD instructions that have not yet passed their WB stage as alternative inputs to the ID/OF stage. However, the result of a LOAD instruction is only available at the end of the MA stage, which is two classic and even three OpenProcessor pipeline stages away from the ID/OF stage. The result of a LOAD instruction is thus not yet available when the following instruction leaves its ID/OF stage, hence it cannot be forwarded in time. As a consequence, this following instruction either experiences a *load delay* [20] and reads the old register value, or must itself be delayed for one cycle in the classic pipeline and two cycles in our system – despite full forwarding logic. On the bright side, this case only costs a little performance but does not increase hardware complexity as no additional/different forwarding paths are required this time.

The alternative to stalling, i.e., the introduction of two load delay slots, so that a LOAD will – by definition – only be visible at the third instruction after the LOAD instruction, has been rejected due to its awkward assembly level programming model and due to the fact that exceptions and interrupts would interfere badly with these load delay slots, making them hardly usable.

Misses in either the instruction cache or the data cache cause memory accesses to take more than the two cycles allotted to them in the pipeline. In these cases, we halt the complete pipeline and wait for the cache to fetch the required data from main memory. On instruction cache misses, the later stages could continue execution and fill in no-operation bubbles from the stalled IF stage so that more potential operands could be computed early and thus later stalls be avoided, but as the benefits are considered small compared to the increase in pipeline control complexity, we stick to the simpler scheme for now.

2.1.5 Two Modes of Execution

The OpenProcessor core supports both a typically restricted user mode of execution and a fully privileged kernel mode. The two modes can be entered using two branch instructions: SYSCALL will (re-)enter kernel mode, while RTU will "return" to user mode. Apart from the actions taken on mode switches (see below), the two modes are not fundamentally different from the hardware's point of view: The mode switch merely serves as a trigger for loading a new set of permissions from a configuration register (R46/R47 for kernel/user mode) as well as a new address space identifier (ASID) from another register (R44/R45 respectively). After the mode switch has been completed (i.e., only after its branch delay slot has been executed using the previous mode), all memory references to both instruction and data memory will be governed by the new ASID, whereas access to privileged resources such as memory mapped TLB configuration registers or the privileged CPU registers R32–R47 will be controlled by the newly loaded configuration register, whose contents is shown in Table 3. This configuration register is designed so that it is safe to use an initial value of 0 to start up in a fully privileged mode with interrupts and address translation disabled, so that low level system startup code can ignore IRQs and TLB management issues.

Bit	Effect when set
0	Enable interrupt processing.
1	Enable virtual addressing (enable TLB).
2–29	(unused)
30	Disable access to TLB control registers.
31	Disable access to privileged CPU registers.

Table 3: The configuration registers r46 and r47 control access to privileged resources such as TLB configuration registers and privileged CPU registers separately for the two modes of execution.

The hardware switches to kernel mode implicitly on interrupts and exceptions or explicitly via a SYSCALL instruction. Execution in kernel mode always starts at one of twelve hard-wired entry points: Six of them are used when entering kernel mode from user mode, the other six are used when the CPU was already in kernel mode prior to the kernel-mode (re-)entry, which allows the kernel, e.g., to load a new kernel stack on kernel entries but continue execution on the current kernel stack during re-entries. The six entry points reflect the cause of the kernel entry: system startup, interrupt, I-TLB miss, instruction decode exception (privileged register requested without permission), D-TLB miss, or system call. For security reasons, even system calls are vectored through a single entry point rather than allowing the user to specify an entry point in the kernel code.

On implicit kernel-mode entries, the hardware saves the interrupted CPU state:

- the contents of the STATUS register after the last successfully executed instruction in R32,
- the instruction pointer of the first instruction that has *not* been executed to completion in R33,
- and the instruction pointer of the instruction to execute after the one pointed to by R33 in R34.

While R34 will in most cases be 4+R33, it is still required to correctly continue execution when an instruction in a branch delay slot has been interrupted or caused an exception. In this case, R33 will point to the branch delay slot, whereas R34 will point to the target of the (taken) branch.

On explicit kernel entries via the SYSCALL instruction, no state is saved in the special purpose registers. Instead, the return address is stored in the link register (R31) as for regular CALL instructions.

In either case, returning to user mode is done using the RTU instruction, which is a regular register-indirect branch instruction that takes the destination address from its register argument (typically R33). After having executed its branch delay slot, RTU enters user mode by loading the user-mode ASID and configuration from R45 and R47 and sets up the instruction fetch logic to continue with the instruction pointed to by R34. Additionally, the first user-mode instruction is tagged to use the saved STATUS register from R32 instead of the current STATUS register. Finally, immediate re-entry to kernel mode must retain the continuation address in R34, which is also achieved by tagging the first instruction in user mode.

2.2 Virtual Memory Management

The OpenProcessor platform provides virtual memory by means of a software controlled, 4-way TLB to map 4 KiB pages. The TLB can be configured at synthesis time to offer either a dual-ported unified TLB with a single buffer per way for 512 translation entries that is shared among the instruction and data paths, or to offer split instruction and data translation buffers for 256 translation entries each (per way).

2.2.1 Implementation

Each of the TLB ways is implemented using two of the dual-ported 2 KiB RAM blocks embedded in the Virtex 4 FPGA, which also explains the size of 512 entries: as detailed below, 4 byte per RAM block are used per translation entry, so that 512 entries make the best use of the 2 KiB RAM blocks.

With 4 KiB pages (12 bit offset) and 256/512 entries (8/9 bit TLB set index), we need to store 12/11 bits of the incoming addresses (the tag) in one of the RAM blocks to facilitate proper hit detection. To discern addresses from different address spaces, we store the address tag together with the 20 bit address space identifier (ASID) in the first RAM block. The other RAM block stores the physical frame number to which the page is mapped (upper 20 bit of the physical address) and up to 12 access permission bits representing CPU-defined access types: right now these are *valid*, *read*, *write*, and *execute*. A TLB hit is reported only if a translation entry matching the requested virtual address and ASID is found and has at least all the permission bits set that are requested by the CPU (which always includes the valid bit). Additionally, one of the permission bits is used to indicate that the page may be cached – which should be clear for memory mapped I/O pages.

The first access port of the RAM blocks is used to fulfill instruction TLB accesses, the second port is used to serve data TLB requests. In the split configuration, the instruction TLB covers the first 256 entries of the RAM block – the data TLB covers the upper 256 entries. This scheme allows the data path to access the instruction path for updates, as required below.

2.2.2 Interface

Differing from MIPS and many other architectures, there are no instructions or CPU registers dedicated to TLB management. Instead, the TLB offers its configuration interface, consisting of four registers to accept the virtual page number, the physical frame number, the address space identifier and the permission bits for a new address translation entry, mapped to *virtual* memory: If one of the virtual addresses assigned to TLB configuration is written to, the TLB will check if the current thread is allowed to access the TLB configuration (see configuration registers R46/47, Table 3) and either update the addressed TLB configuration register or try to map the address to its physical address. When the virtual address register is updated, the values from the configuration registers are converted to TLB entry form and committed to the translation buffer. The ways that are to be affected by such an update are conveyed as a bitmask in the (currently) four least significant bits of the virtual address to map, so the software is in full control of which TLB way(s) is/are to be used. This scheme allows to quickly flush the TLB by invalidating all four ways of a set at once, to reserve TLB ways for certain applications, or even to simulate a smaller TLB by leaving 1–3 ways unused.

The memory mapped approach to managing TLB entries provides a very flexible interface, which requires only few changes to the rest of the hardware system when changed. On the downside, updates to the instruction TLB cannot be handled that way, as no writing memory access ever occurs on the instruction path. To work around this problem, we merge the instruction and data TLBs into one module with a shared buffer space and allow the data path to update translations for both the data and the instruction path. For this purpose, the virtual address register is mapped to two memory locations: one serves to update the data path translations, the other triggers updates to the instruction path. For unified TLB configurations, both register accesses behave in the same way and predictably update the (shared) translation entries. Software does not even have to know if the TLB is currently configured to be split or unified; the software can always access the TLB as if it were split between data and instruction path, which facilitates comparable measurements with the only modification between test runs being the TLB configuration.

On TLB misses, the hardware raises a TLB miss exception, indicating which way best matched the requested translation (to allow adding permissions) and which access type was requested by passing the up to 12 access permission bits from the CPU on to the exception handler in R36. There is currently no hardware-support for clever TLB entry replacement (e.g., based on time since last use or access frequency), so that we employ a global round robin TLB way assignment and replacement scheme. However, future implementations are free to provide more relevant information to the handler to improve the system in this respect.

2.2.3 *Previous Approach*

The TLB configuration presented above is the second one that we implemented for the platform. Initially, we had implemented a more flexible scheme which allowed us to map arbitrary 2^n-sized pages by storing an address mask along with the virtual and physical addresses. All addresses that matched the stored virtual address when masked with the stored address mask were mapped to the stored physical addresses after having ORed in the unmasked bits of the incoming addresses (typically an offset). This scheme required parallel checks on all the TLB entries (fully associative), which in turn ruled out the RAM blocks as storage, so that we had to revert to store the translation entries in registers within the FPGA fabric. As each of the translation entries (including the address mask) required $3 \times 32 = 96$ bits plus permission and valid bits, the number of TLB entries to be implemented that way while keeping the hardware acceptably small seemed to be too small to be of much use: We started with four entries, which proved to be too small to accommodate mappings for both kernel and user mode, and thus increased the number to eight TLB entries before switching to the less flexible, but simpler and more maintenance friendly scheme presented above. More research is required to determine sane limits and evaluate benefits of the flexible approach.

2.3 I/O Devices

As the development board provides only 64 MiB of DDR RAM plus 4 MiB of Flash, the 32 bit physical address space of the OpenProcessor platform offers enough space to cover any I/O devices' control, status, and data registers including device buffers (e.g., for the Ethernet controller). Thus, all I/O devices are accessed via memory mapped I/O, the address ranges and relative locations of the device-specific registers are defined manually at synthesis time and inserted into a (non-machine readable) memory map to avoid conflicts and overlaps.

2.3.1 *Memory Controllers*

The most important device on the platform is the SRAM controller, which serves to make up to 32 FPGA internal RAM blocks available as memory via the system bus. This memory is initialized at synthesis time with a bootloader to start up the system.

Other memory-related devices on the platform include the TLBs and the caches, the latter of which only expose a single control register that can be written to in order to write back and invalidate the whole cache (TLB management has been discussed in Section 2.2). Similarly, the DDR memory controller can be configured at runtime with respect to its CAS latency and with respect to the phase shift of the external DDR clock relative to the system clock, which has been an important feature while debugging DDR memory access.

2.3.2 *Programmable Timer*

Aside from memory, there is a programmable timer that can be used to either generate interrupt requests at a fixed rate (anywhere between every and every $2^{32} - 1$ cycles) or only once after up to 2^{32} cycles. This device is used by the operating system to facilitate round robin scheduling by setting up a one-shot timer event to occur after the remaining timeslice length of the scheduled thread has expired.

2.3.3 *Basic I/O*

The user I/O devices (eight DIP switches, four pushbuttons, and four LEDs) are controlled by a general purpose I/O controller module that provides the status of the switches and buttons via its status register and takes a four bit value per LED at its configuration register, which is used to control the brightness of the LEDs via pulse width modulation (PWM). For convenience, the current LED status is also readable from the status register to facilitate read-modify-write operations and relieve the software from having to keep track of this state, which would be required if one wants to change only one of the LEDs. The LEDs have provided a valuable debugging aid in the early stages of the system when serial I/O or even network was not yet/no longer possible and continue to do so, e.g., by signaling the current mode of execution/pending interrupts/thread switches/...

The second important debugging aid is the liquid crystal display (LCD), which provides two lines of 16 characters each. The LCD character buffer is mapped to physical memory, so that displaying text is as simple as writing the text to the character buffer. This display is particularly helpful in situations where the system freezes, as the display will retain the last text written there before the freeze. As access to the LCD is significantly faster than writing the characters to the PC via RS-232 (which is the third means of debug output, but limited by the PC's maximum baud rate to no more than 1 Mbaud or ≈ 100 kB/s as compared to several MB/s on the LCD), the LCD can be used to quickly locate the problem. More debug output (exceeding the 2x16 characters the LCD can handle) can then be emitted via RS-232.

The RS-232 controller is the third I/O device in the system. It exposes a status word that indicates whether a character is available, which character this is, and whether the transmit buffer is full. For efficiency, the RS-232 module internally maintains a receive FIFO buffer as well as a transmit FIFO buffer of configurable sizes (currently 8 entries for both); however, the buffers can only be accessed in FIFO order by reading a single character from the data register or by writing a single character to the transmission control register of the controller.

2.3.4 Ethernet

For high-speed communication, the OpenProcessor platform provides a 100 Mbit Ethernet controller, which internally uses an 8 KiB receive buffer to buffer incoming packets and a 2 KiB transmit buffer, which suffices to hold a single Ethernet frame (which is at most 1522 bytes long as per the IEEE 802.3 Ethernet standard). Both buffers are mapped to memory in the same 8 KiB region because the receive buffer is read-only and the transmit buffer is write-only.

A 32 bit status register indicates whether a (new) packet has been received, including its length (11 bit) and position (13 bit) in the receive buffer. By reading this status register, the software also indicates that it has completed processing the *previous* packet, so that the receive logic can reuse its buffer space. Similarly, the Ethernet module provides a status register that conveys whether the module is idle or (still) busy sending the previous frame. If it is idle, the software can copy the next frame to the transmit buffer and trigger the transmit logic by writing a 16 bit start offset and a 16 bit length to a control register.

The Ethernet module can be configured to provide basic Ethernet protocol off-loading services such as generating and stripping off the Ethernet preamble that precedes all frames. Furthermore, the module can generate and check the frame check sequence (FCS, a CRC-32) at the end of an Ethernet frame to ascertain data integrity; incoming frames with an invalid FCS are then discarded.

As a specialty, the Ethernet module makes up for the misalignment of Ethernet payload caused by the 14 byte long Ethernet header. In order to align payload data to 4 byte boundaries, the receive logic starts writing Ethernet frames at addresses equal to 2 (mod 4) into the receive buffer to facilitate efficient, word-wise access at higher protocol layers. Similarly, the Ethernet module can start sending from arbitrary addresses within the transmit buffer, so that outgoing data can be copied in word-wise, possibly including invalid data just before the frame, which can then be skipped by starting transmission at the proper offset. Fixing alignment in the Ethernet module is important, as byte-wise copying or access to unaligned 32 bit words (implemented by GCC using four byte-accesses) takes four times as long as word-wise, aligned accesses – a significant factor considering the large amount of data to be processed by the network stack.

2.3.5 Interrupts

Notifications from the I/O devices to the CPU can be triggered using interrupt request lines per device. However, the interrupts are not delivered from the source devices directly to the CPU; instead they are aggregated and mediated by a programmable interrupt controller (PIC). The PIC can be configured to mask selected interrupt sources to enable polling (or ignoring) them and to protect the user-mode handler from nested requests. The PIC can also be queried to determine which interrupts are pending (even if masked).

At the moment, there are four possible interrupt sources: each of the four pushbuttons asserts an IRQ while pressed, the timer module asserts an IRQ if the timer expires, and both the RS-232 and the Ethernet module indicate idle transmit states as well as the availability of new data using two separate IRQ lines each.

3. MONITORING AND DEBUGGING

Being designed with future improvements in mind, the OpenProcessor platform provides a number of features dedicated to performance measurements/monitoring and low-level debugging.

3.1 Hardware Event Counters

The most fundamental means to assess hardware performance is to count the number of cycles a specific operation requires. The OpenProcessor platform facilitates this by exposing a number of 64 bit counters to provide basic execution statistics. All 64 bit counters in this section are accessed by reading the lower 32 bit from their respective base register (R49 to R63), which causes the upper 32 bit to be latched into the HOLD register (R48), whence they can be retrieved at a later time.

Specifically, the platform publishes a timestamp counter that counts clock cycles since startup in R63. Together with R61, which reveals the number of instructions executed since startup, this can be used to compute the number of cycles per instruction (CPI), which should ideally be close to 1 on this platform.

If the CPI value is significantly larger than 1, other registers can help to identify the cause: R62 counts the number of stall cycles of the pipeline, i.e., usually cycles spent waiting for memory. The remaining "unproductive" cycles ($\text{R}63 - \text{R}61 - \text{R}62$) are wasted on no-operation cycles in the pipeline, introduced by pipeline flushes due to interrupts and exceptions and by taken branches, the number of which is accessible via R60.

More details on the cause of stall cycles can be obtained from R56 through R59, which respectively provide the number of stall cycles called for by the ID/OF stage (waiting for memory operands that are provided too late), the number of TLB-related stall cycles (caused by updates to the translation entries, as each one causes a one cycle delay), and the number of cycles the instruction and data caches have waited for data from memory.

For short code fragments, it is sufficient to consider the difference of only the low 32 bit of these counters, which keeps the overhead negligible if only few counters are evaluated. If the counter values are to be recorded at a high frequency or over a longer period of time to facilitate "side-effect free" offline analysis, however, reading the counters and especially writing them to memory has been seen to seriously impact performance: Measurements of the cycles used by the interprocess communication code path went up by nearly 50 % (from 11,000 to over 20,000 cycles) just by adding instrumentation along the path. Thus, we provide a better alternative for this purpose: the software event log.

3.2 Software Event Log

Instead of reading the counters and writing them to memory in software, we provide a special EVENT instruction that accepts two register arguments and a 10 bit immediate and logs them together with a selection of the previously mentioned counters to a dedicated circular buffer that provides room for 2048 entries of 8 words each. The words to log are configured at synthesis time and usually include the two operands to the EVENT instruction, the opcode (to gain access to the immediate), the address of the EVENT instruction to map the event to the code, and the low words of the timestamp counter, the stall cycle counter, the executed instructions counter and the taken branches counter.

The log is mapped to memory as an array of 8 words per entry, so that it can easily be read once the instrumented code path has been executed. Despite being implemented as a circular buffer, the first record (at offset 0) always maps to the most recent log entry to simplify log evaluation. The last entry in the array (index 2047) thus always represents the oldest log record available (if any).

Using the EVENT instruction instead of manually writing the counters to memory both reduces code size (one instruction instead of eight to store eight counter values – not counting target address computations) and runtime overhead especially by avoiding cache misses in the recording phase.

The downside of this approach is that is is pretty costly in terms of hardware resources: 2048 records of 8×4 bytes each amount to 64 KiB of memory or 32 RAM blocks of 2 KiB each. Considering that only 160 such RAM blocks are available in the FPGA, such logs are expensive and cannot grow much larger than that.

3.3 CPU Trace Unit

During debugging of software that executes on the OpenProcessor platform, we often found it desirable to be able to go back in time to see where an observable error such as a pagefault at an invalid address was caused. Using the logs described above, we provide a configurable trace unit that logs the address, opcode, both register operands and ALU result (if any) plus ASID, pipeline control bits, the 16 least significant bits of the timestamp counter and the contents of the STATUS register of the last 2048 instructions executed for later inspection.

The trace unit can be configured to trace execution in user mode and/or in kernel mode, it can be restricted to log execution only in a given address space or it can be disabled to preserve its current state. Typically, we setup the trace unit to trace execution of both user and kernel mode, but turn it off once we have reached kernel mode and turn it on again just before we return to the user mode of an "interesting" address space. This provides us not only with the trace of the potentially faulty execution in user mode, but also conveys register values that are saved during kernel entry just before the trace unit is turned off – a valuable aid in debugging.

However, the trace unit is not limited to debugging purposes: The inclusion of the least significant 16 bits of the timestamp counter facilitate detection of pipeline conflicts in the past execution (indicated by adjacent instructions with non-adjacent timestamps) or unexpected cache misses.

3.4 Cache Activity Log

To further analyze cache behavior, we employ another log (using only three instead of eight rows per entry to save memory) to record a history of cache misses, cache hits, write back and/or flush events. The cache log can even be configured to record the data values that are loaded into or read from the cache. However, it turns out that this hardly ever conveys useful information not yet available from the CPU trace log.

In addition to the cache activity log, the cache implementation itself provides access to a number of 64 bit counters to count hit, miss, and write back events of the cache. Furthermore, the cache also counts the number of cycles spent on writing cache lines back to memory, on reading data from memory, on flushing the cache, on marking cache lines as dirty (this takes one cycle per previously clean cache line), and even the number of cycles spent on memory accesses that bypass the cache (such as I/O accesses). As with the CPU statistics, the 64 bit values are conveyed by the cache using a 32 bit hold register in addition to the counter-specific base registers, all of which are mapped to memory addresses.

4. KAOS

The OpenProcessor platform provides system software in addition to the synthesizable hardware, not only to test and demonstrate the hardware, but also to provide a starting point for the co-development of hardware and software features, such as the page replacement algorithm based or page access timestamps or counters as motivated in the introduction.

Due to the custom instruction set of the OpenProcessor core, we first needed to provide a cross-compiler toolchain for the platform. Given some prior knowledge of the GNU GCC and binutils, we implemented a rudimentary back-end to enable GCC and the GNU binutils to generate code for the OpenProcessor core. This compiler back-end is incomplete (support for 64 bit data types, C++ exceptions, debugging information and floating point are all missing), so we decided

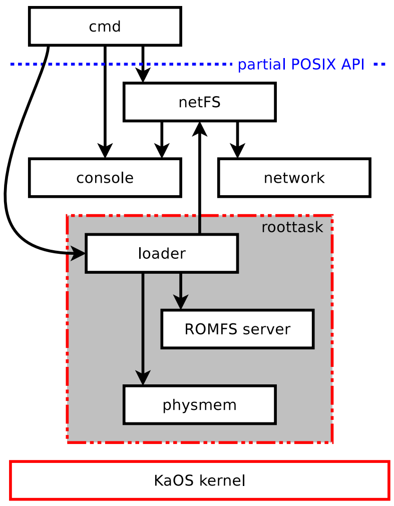

Figure 3: The OpenProcessor includes KaOS, a microkernel-based multi-server operating system that provides a subset of the POSIX environment for applications.
The KaOS kernel and the roottask are trusted components; arrows indicate the "calls" relation.

not to try to port an existing OS such as L4, Minix 3, or Linux to the platform to start with. Instead, we went for an incrementally co-developed microkernel-based system that – being completely understood – would simplify debugging. The system is depicted in Figure 3.

4.1 KaOS Kernel

The KaOS kernel is a microkernel heavily inspired by L4 [10], but implemented in C (not C++). Both kernels provide threads and address spaces as their sole abstractions, complemented by synchronous IPC (with timeouts) to interact with threads and a virtual memory mapping mechanism to recursively construct address spaces from user mode. Notable differences from the L4 kernel are the absence of kernel-provided memory structures such as L4's kernel interface page or user-level thread control blocks, since these are primarily provided to support portability of the L4 kernel and are thus not required in a system that targets a single architecture.

The KaOS kernel handles exceptions, delivers interrupts via IPC to registered interrupt handlers, creates, destroys, and schedules threads using a priority-based round-robin scheduler, and updates TLB entries after TLB misses.

4.2 Support Servers

On top of the KaOS kernel, a number of applications (so-called *servers*) implement a subset of the POSIX API with the notable exception of forking, which is difficult in microkernel-based systems due to the distributed state of its processes: Applications typically hold handles on many servers, and duplicating all of them on fork is often too heavyweight. Instead, we replaced forking by a combined fork-and-exec routine to execute programs in a new address space (similar to Windows' CreateProcess) and a lightweight KaOS system call to create a new thread.

Three servers provide the basic functionality. The remaining servers are loaded as applications from a ROMFS image (similar to Linux' initial ramdisk), which is facilitated by a *ROMFS server* that offers access to the filesystem using standard `open`/`read`/`close` calls. Loading applications is controlled by a *loader* server that also manages the address space layout of the applications (with respect to thread's stacks and memory mapped files) and serves as their pager, meaning that the kernel will ask the loader thread to resolve page faults if the kernel cannot resolve TLB misses on its own. The memory that the loader hands out to its clients on pagefaults comes from *physmem*, a server that manages free physical memory for use by pagers such as the loader.

Once these servers are up and running, the system can load additional servers (e.g., from the ROMFS) to enhance functionality: Most importantly, a console server can be used to synchronize output to the serial console and to give all threads `stdout`, `stderr`, and `stdin` streams without having to grant them access to the RS-232 hardware directly.

Another central server provides network functionality via Berkeley sockets together with Ethernet/IP/UDP layers and low-level network services such as BOOTP, (R)ARP and ICMP. This network server is then used to grant access to a network filesystem, implemented via an augmented TFTP server that supports partial read and write requests to access only selected parts of a file, directory listings and a `stat` command to obtain information about the type, size, and timestamps of a remotely available file.

4.3 C Library

The functionality of the servers is offered either using a server-specific remote procedure call (RPC) interface, or hidden behind the standard libc/POSIX calls. While the former interface is slightly more efficient, it is not well portable and often awkward to use. The libc/POSIX interface, on the other hand, must typically translate file handles to (serverId, handle) tuples and can then use the server-specific interface to provide the desired functionality.

Due to this translation requirements, we implemented a new libc for the OpenProcessor platform, which can (statically) be linked into the servers and applications to support their implementation. As a specialty, most of the libc can be used in a way that does not involve any support servers, which allows even the KaOS kernel to use the provided `printf` for output via the RS-232 module and `malloc` and `free` for memory management. However, `malloc` usually uses `mmap` to obtain or enlarge its managed heap, but since `mmap` is provided by the loader server, this approach is not viable for the kernel. As a workaround, the kernel provides a predeclared heap (currently 256 KiB) at a non-zero address conveyed to the library through the symbol `__heap_start`. Regular applications define this symbol to `NULL`, thus indicating that `mmap` can and should be used.

4.4 System Startup

The OpenProcessor platform consists of both hardware and software, so the software could be integrated into the hardware by employing initialized RAM blocks within the FPGA.

However, hardware synthesis takes about 25 minutes on a 1 GHz Intel Core Duo with 1 GiB of RAM, whereas software compilation takes less than 30 seconds on the same machine. With software being expected to change more often than hardware, we need to separate the software from the hardware and combine them only at runtime – possibly updating the software several times without having to rebuild the hardware.

One way to solve this problem is to start execution after startup from persistent memory such as the 4 MiB of Flash on the development board. Due to its limited space and awkward programmability from the host PC, we take a different approach and load the software from the PC every time after startup using a stable piece of software (4 KiB code plus 1 KiB data) that is embedded into the FPGA RAM blocks that are mapped at addresses 0–64 KiB. This SRECLoader is considered "stable" in the sense that it does not change regularly, so embedding it into the hardware poses no problem.

After system startup, execution is directed to the SRECLoader program. The SRECLoader waits for data to be pushed onto the FPGA board via RS-232 in the form of S-records, i.e., hex-encoded chunks of up to 16 bytes of data per line preceded by a target address and protected against transmission errors using a simple checksum. Upon reception of a line, the SRECLoader validates the checksum, decodes and writes the data into place and sends an acknowledgment (or error indication) back to the PC, which then either retransmits the faulty line or continues with the next S-record. A special type of S-records provides the address of the entry point into the loaded code (if any) and also serves to terminate the loading process.

The SRECLoader could be (and has been) used to directly load the KaOS kernel and its support servers into memory and start their execution. However, being as simple as it is, the loader transfers data at only about 4 KiB/s from the PC; loading the 400 KiB kernel, the 450 KiB roottask that includes the three basic servers (cf. Section 4.2), and the 250 KiB ROMFS image, which provides the remaining system servers, would take more than 4 minutes. Instead, we use the SRECLoader to load a 9 KiB ELFLoader, which in turn takes less than 20 s @ 60 KiB/s via RS-232 to load the KaOS kernel, the roottask, and the ROMFS image by using larger chunks and by avoiding to hex-encode the data for transmission, which doubles its size.

After having loaded the three modules, the ELFLoader creates and stores a memory map (in memory), in order to convey the type, size, and position of the loaded modules to the KaOS system. Only after the memory map has been created does the ELFLoader identify the kernel module and start its execution at its entry point, storing a pointer to the memory map in R30 for the kernel to use.

From now on, the kernel initializes its data structures (mainly the thread control blocks) and starts execution of the first user-mode thread – the trusted roottask. For this to work, both the kernel and the roottask must be mapped 1:1 (i.e., their physical addresses match their virtual addresses), because he roottask does not have a pager assigned that could arbitrarily remap its pages. As a consequence, kernel and roottask must be linked to disjoint address ranges.

When starting the roottask, the kernel passes along a pointer to the memory map, so that the roottask can (a) create a list of free memory regions for the *physmem* server to manage, and (b) point the ROMFS server to the start of the ROMFS image, so that the loader service can start the remaining system services.

On startup, the system typically executes a number of tests to continuously validate that both hard- and software are still intact, before dropping into a command line interpreter (calling it a "shell" would be too much at the moment) that allows to interactively load and execute additional programs from the ROMFS image or via network.

4.5 Monitoring and Debugging

In order to profit from the monitoring and debugging capabilities of the hardware, KaOS includes a kernel debugger, which can be invoked at any time by sending a `^` character via RS-232 to the system. Alternatively, the debugger can be invoked through software, e.g., after having detected a problem, using a microkernel system call.

Upon entry, the debugger will dump basic statistics such as the current time stamp, the aggregated time spent in user mode respectively kernel mode, the number of kernel entries and their causes (interrupt, exception, system call) and the number of thread switches. To facilitate rough but simple measurements of these figures for the code that executes between two invocations of the debugger, the differences of these figures from their values at the previous kernel debugger invocation are also output on entry to the debugger.

After that, the debugger provides a command line interface, which allows to enter a number of commands to inspect the states of threads and address spaces in the system, to dump the event log, CPU trace, and cache log, to list pending IPC timeouts (the only form of application-specific timers managed by the kernel), and even to inspect hexdumps of memory regions of arbitrary address spaces.

While the system executes the kernel debugger, the system is frozen: No interrupts are processed, and no scheduling decisions are taken to dispatch ready-to-run threads until the kernel debugger is left using the `quit` command.

5. USE CASES

The OpenProcessor has already been used to conduct research in two major areas: Firstly, its memory access path has been augmented to better support page replacement decisions, and secondly, the system has been augmented to keep the contexts of multiple threads in hardware to facilitate fast switching between interacting threads.

5.1 Memory Profiling

One of the motivations to develop the OpenProcessor platform was to improve the page replacement decision of operating systems by providing more information per physical page than the single referenced bit: access counters to support least frequently used (LFU) page replacement, or timestamps to support the (generally favored) least recently used (LRU) policy.

In [4] we have examined the feasibility of reference counting on the OpenProcessor platform. Considering that maintaining the counters should not interfere with regular system operation, the counters have to be placed into a dedicated memory separate from the system's main memory; placing them in some of the FPGA internal RAM blocks is the option of choice.

However, even with only 64 MiB of main memory on the development board, using a standard page size of 4 KiB yields 64 MiB / 4 KiB = 16 k physical pages and providing a counter of 32 bit per physical page would already require a total of 64 KiB of memory or 32 RAM blocks of 2 KiB each. As RAM blocks are typically one of the scarcest resources on an FPGA, we opted to only monitor a configurable region of memory at any time.

To this end, we provide four memory profiling units, each of which can monitor 512 successive blocks of memory of 2^n bytes within a naturally aligned, contiguous region of 2^{9+n} bytes with n and the starting address being configurable at runtime. The counter values are 32-bit wide (so that each memory profiling unit uses a single 2 KiB RAM block), are read/write accessible at runtime and can thus be used by the OS to drive its LFU page replacement strategy.

Further work conducted after [4] added configuration options to make the reference counters saturate instead of wrap around at $2^{32} - 2$, and to disable individual counters by writing $2^{32} - 1$ into them to support "pinned" pages (pages that should not be replaced).

As retrieving the 4×512 counters from all the memory profiling units for each page replacement decision is both costly and repetitive, [12] added hardware to the OpenProcessor platform that automatically iterates over the counters and remembers the indices and values of the 4 (later extended to 8) smallest counters per memory profiling unit, thus granting the OS immediate access to the 8 best candidates for replacement according to the "least frequently used" (LFU) policy.

Again, further work on the memory profiling units now allows us to record the timestamp of the last access per page instead of counting the accesses using another runtime configuration option to the profiling units. Using the same selection mechanism as above, the four smallest "counter" values (timestamps) now lead to the least recently accessed pages, thus facilitating "least recently used" (LRU) page replacement policies within the monitored memory region.

An evaluation of this last system, including proper handling of timestamp overflows via "aging" is in progress [6].

5.2 Register Banking

The OpenProcessor core is a RISC-based CPU with 32 registers. On each thread switch, all of these registers must be saved to memory and filled with the state of the next thread from memory. Even worse, on each involuntary kernel entry (exception, interrupt), a considerable subset of these registers (namely those that are to be saved by the caller of a function – the caller-saved registers) must be saved to protect them from changes by kernel code.

To reduce the cost of context switches and kernel entries, [14] and [17] investigate the benefits of keeping multiple contexts in hardware by providing multiple register banks, each of which can freely be assigned to a thread's user-mode or kernel-mode context. By assigning two banks to each thread, one to its user-mode execution and one for its kernel-mode execution, kernel entries come at virtually no cost. By dedicating banks to frequently used service threads (pagefault handler, network and file system servers, ...), communication with these services can be improved.

An evaluation of the effects and problems such as access to other thread's register state during IPC or sane bank assignment strategies is in progress [17].

6. RELATED WORK

The OpenProcessor platform covers both reconfigurable hardware and microkernel-based operating systems, so we relate it to a limited set of previous works in both areas separately.

6.1 FPGA-Based Systems

Other systems that use FPGAs to facilitate modifications to a given hardware platform exist. Most prominently, the BEE3 platform [5], intended for system architecture research with a focus on multi-core architectures, is a powerful platform housing multiple Virtex 5 FPGAs and several GiB of memory on custom built PCBs, assembled into standard server racks. Though the platform is used successfully by a number of research projects such as RAMP [18], it is rather expensive and complex and not targeted at the interested hobbyist developer as the OpenProcessor platform is.

ABACUS [11] employs FPGA technology to facilitate fast program profiling by monitoring and analyzing most if not all memory accesses in real time and providing statistical results of memory access performance and bottlenecks. In a way, this is close and even beyond one of the intended research topics for which we developed the OpenProcessor platform, but ABACUS is specialized (and limited) to memory access monitoring, whereas we aim at providing a general research platform for system architecture.

There are also many processor cores available for synthesis and instantiation on an FPGA. Most notably among these are probably the free OpenSPARC T1/T2 [3], multi-threaded open-source implementations of the 64 bit SPARC v9 architecture provided and supported by Oracle (formerly Sun Microsystems), and the community effort OpenRISC, a 32 or 64 bit processor with optional floating point support that is hosted on opencores.org [2], together with a large amount of additional free hardware source code. There are even commercial processor cores available for licensing, e.g., the LEON family of SPARC v8 implementations (see [21] for an overview), or Xilinx' MicroBlaze processor [22], a resource efficient 32 bit processor especially designed to be implemented on Xilinx FPGAs.

However, most of the cores above are either too large and complex for our aims, target improvements of a single given processor architecture, or are too much geared towards a prescribed system architecture to leave much room for modifications in the areas that interest us most: virtual memory management and context/thread management by the operating system.

6.2 Microkernel-Based Systems

The OpenProcessor KaOS kernel borrows heavily from the L4 microkernel [10], especially with respect to the memory mapping techniques and the IPC interface. Projects such as L4VFS [16] that implement a subset of the POSIX API on top of L4 seem to have ceased to exist.

MINIX 3 [9] is a well-known microkernel-based operating system that implements the POSIX API and focuses on robustness. It is more mature and more thoroughly designed than the the KaOS system, and available for the Intel x86 platform – ports to other architectures are planned.

7. CONCLUSION

We have presented the OpenProcessor platform, a fully functional yet relatively simple computing platform that comprises both a customizable hardware basis (implemented in the Verilog HDL) and a microkernel-based operating system that provides a subset of the POSIX API through a number of server applications.

The system is designed to foster research on the hardware and is thus expected to be in constant flux with respect to its interfaces and capabilities. As the system is small and simple enough to be instantiated on inexpensive, off-the-shelf FPGA development boards, it is ideally suited as a basis for hobbyist developers or students. Some student projects have already confirmed that significant contributions can be made in less than six months.

The source code for the complete OpenProcessor platform is available upon request from the authors of this report.

8. REFERENCES

[1] GNU compiler collection. `http://gcc.gnu.org/` (2010-07-13).

[2] OpenCores website. `http://www.opencores.org/` (2010-07-13).

[3] OpenSPARC website. `http://www.opensparc.net/` (2010-07-13).

[4] B. Ahues. Entwurf und Implementierung einer erweiterten Speicherkontrolleinheit. Study thesis, University of Karlsruhe, Aug. 2008. `http://os.ibds.kit.edu/97_357.php` (2011-01-14).

[5] J. D. Davis, C. P. Thacker, and C. Chang. BEE3: Revitalizing computer architecture research. Technical Report MSR-TR-2009-45, Microsoft Research, Apr. 2009.

[6] S. Friedmann. Benchmark-based evaluation of hardware-assisted page replacement strategies. Study thesis, Karlsruhe Institute of Technology (KIT), Mar. 2011. `http://os.ibds.kit.edu/97.php` (to appear).

[7] J. R. Goldman. Coherency for multiprocessor virtual caches. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 72–81, 1987.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufmann, second edition, 1996.

[9] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Reorganizing UNIX for reliability. In C. Jesshope and C. Egan, editors, *Proceedings of the Eleventh Asia-Pacific Conference (ACSAC)*, volume 4168 of *Lecture Notes in Computer Science*, pages 81–94. Springer, Sept. 2006.

[10] J. Liedtke. Towards real microkernels. In *Communications of the ACM*, volume 39, pages 70–77, Sept. 1996.

[11] E. Matthews, L. Shannon, S. Blagodurov, S. Zhuravlev, and A. Fedorova. ABACUS: A hardware-based software profiler for modern processors. *Poster and demo at the Fifth European Conference on Computer Systems (EuroSys)*, 2010.

[12] S. Müller. Improving memory management with hardware-generated memory access profiles. Study thesis, University of Karlsruhe, June 2009. `http://os.ibds.kit.edu/97_1385.php` (2011-01-14).

[13] OpenCores Organization. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, Sept. 2002.

[14] S. Ottlik. Reducing overhead in microkernel based multiserver operating systems through register banks. Study thesis, Karlsruhe Institute of Technology (KIT), Oct. 2010. `http://os.ibds.kit.edu/97_2194.php` (2011-01-14).

[15] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*, chapter A. Morgan Kaufmann, third edition, Aug. 2004.

[16] M. Pohlack and B. Döbel. L4VFS: L4 virtual file system layer, Sept. 2006. `http://demo.tudos.org/l4vfs_tutorial.html` (2010-07-13).

[17] V. van Santen. Design and implementation of multiple register banks to improve context switch performance on the OpenProcessor platform. Study thesis, Karlsruhe Institute of Technology (KIT), Mar. 2011. `http://os.ibds.kit.edu/97.php` (to appear).

[18] J. Wawrzynek, M. Oskin, C. Kozyrakis, D. Chiou, D. A. Patterson, S.-L. Lu, J. C. Hoe, and K. Asanovic. RAMP: A research accelerator for multiple processors. Technical Report UCB/EECS-2006-158, EECS Department, University of California, Berkeley, Nov. 2006.

[19] Wikipedia. Classic RISC pipeline, 2010. `http://en.wikipedia.org/w/index.php?title=Classic_RISC_pipeline&oldid=364211949` (2010-07-28).

[20] Wikipedia. Delay slot, 2010. `http://en.wikipedia.org/w/index.php?title=Delay_slot&oldid=370853871` (2010-07-28).

[21] Wikipedia. LEON, 2010. `http://en.wikipedia.org/w/index.php?title=LEON&oldid=366625033` (2010-07-28).

[22] Xilinx Inc. The MicroBlaze soft processor core. `http://www.xilinx.com/tools/microblaze.htm` (2010-07-13).