electronics

MDPI

Article

HAL-ASOS Accelerator Model: Evolutive Elasticity by Design

Vítor Silva [1], Paulo Pinto [1], Paulo Cardoso [1], Jorge Cabral [1,2,*] and Adriano Tavares [1]

[1] ALGORITMI Centre, University of Minho, 4800-058 Guimarães, Portugal; vsilva@dei.uminho.pt (V.S.); ppinto@dei.uminho.pt (P.P.); pcardoso@dei.uminho.pt (P.C.); atavares@dei.uminho.pt (A.T.)
[2] CEIIA Centro de Engenharia e Desenvolvimento de Produto, 4550-017 Matosinhos, Portugal
* Correspondence: jcabral@dei.uminho.pt or jorge.cabral@ceiia.com

Abstract: To address the integration of software threads and hardware accelerators into the Linux Operating System (OS) programming models, an accelerator architecture is proposed, based on micro-programmable hardware system calls, which fully export these resources into the Linux OS user-space through a design-specific virtual file system. The proposed HAL-ASOS accelerator model is split into a user-defined Hardware Task and a parameterizable Hardware Kernel with three differentiated transfer channels, aiming to explore distinct BUS technology interfaces and promote the accelerator to a first-class computing unit. This paper focuses on the Hardware Kernel and mainly its microcode control unit, which will leverage the elasticity to naturally evolve with Linux OS through key differentiating capabilities of field programmable gate arrays (FPGAs) when compared to the state of the art. To comply with the evolutive nature of Linux OS, or any Hardware Task incremental features, the proposed model generates page-faults signaling runtime errors that are handled at the kernel level as part of the virtual file system runtime. To evaluate the accelerator model's programmability and its performance, a client-side application based on the AES 128-bit algorithm was implemented. Experiments demonstrate a flexible design approach in terms of hardware and software reconfiguration and significant performance increases consistent with rising processing demands or clock design frequencies.

Keywords: hardware task; hardware accelerator; hardware kernel; FPGA; microcode; dynamic partial reconfiguration; elastic hardware system calls; evolutive elasticity by design

Citation: Silva, V.; Pinto, P.; Cardoso, P.; Cabral, J.; Tavares, A. HAL-ASOS Accelerator Model: Evolutive Elasticity by Design. *Electronics* **2021**, *10*, 2078. https://doi.org/10.3390/electronics10172078

Academic Editor: Jean-Christophe Prévotet

Received: 23 June 2021
Accepted: 23 August 2021
Published: 27 August 2021

1. Introduction

Since today's most frequent demands for embedded devices are still grounded to performance, several operating systems for FPGA have been proposed to tackle the real-time performance issue of the CPU-only computing system, mainly through FPGA's reconfigurability and high energy efficiency. The mix of fast CPU cores and fine-grained reconfigurable logic allows mapping of both sequential or control-dominated code and highly parallel data-centric computations into a single platform. However, the programming models for software and reconfigurable hardware lack commonalities, which, in time, will hinder design space exploration (DSE) and lower the potential for code reuse.

For a better comprehension, Figure 1 shortly presents an overview of the HAL-ASOS accelerator design flow used to generate an application-specific operating system (ASOS), which ensures an efficient design and eases the programmability gap between software and hardware concepts. Furthermore, it provides the designer with a complete solution for developing reconfigurable systems that benefit from the synergy among software, hardware, and services, as well as deliver powerful computation solutions that can be built with just the right and needed resources. Basically, the development process is carried out through the following steps: (1) writing a new application or refactoring an existing application according to a class Task concept; (2) running a parallelization tuning cycle using profilers to identify critical Linux kernel- and user-level subsystems that should be tuned for scalability; (3) running a co-simulation stage by applying the accelerator to the

selected offload task candidates, through the integration of high-level synthesis tools such as Vivado HLS or MATLAB, to translate C/C++ programming models to appropriated register-transfer-level (RTL) representation; and (4) supporting the system designer in the creation of the full platform solutions, including board support package (BSP), stripped bare minimal Linux OS, device drivers, middleware and applications software. However, high-level design methodology will not be addressed in this paper. The main focus goes towards the Hardware Kernel and its microprogrammed control unit, which, compared to the state of the art, will leverage the elasticity to naturally evolve with Linux OS through key differentiating capabilities of field programmable gate arrays (FPGAs).

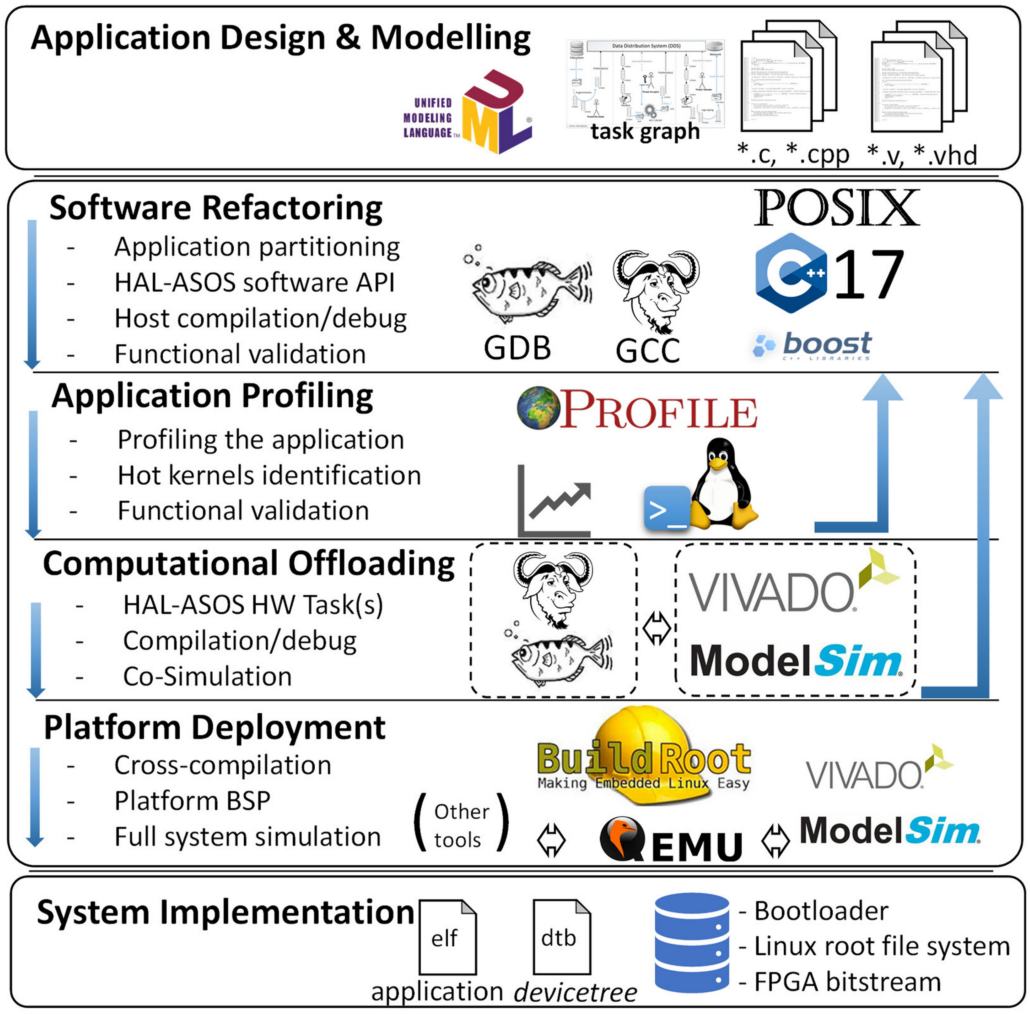

Figure 1. The HAL-ASOS accelerator model design flow.

Figure 2 depicts a simplified representation of a HAL-ASOS accelerator model as split into a user-defined Hardware Task and a parametrizable Hardware Kernel, with the former using the latter to interact with the host system (i.e., the selected hardware platform that includes the CPU, the physical memory, and the Linux OS). The transfer channels are platform-dependent and establish differentiated data exchange with the host system. These include: (1) a fast, word-rated, and low-bandwidth channel used for control-oriented transfers; (2) an optimized speed, a byte-rated and high-bandwidth channel used for large and data-oriented transfers; and (3) a byte-rated and high-bandwidth channel used by Hardware Task to access the system memory. Platform-classified model implementations will include PLB or AXI Bus interfaces. The HAL-ASOS accelerator is a native 32-bit big-endian machine (64-bit word can be applied system-wide) that includes a mandatory

interrupt line to allow the accelerator to synchronize with the Linux OS, as depicted in Figure 2.

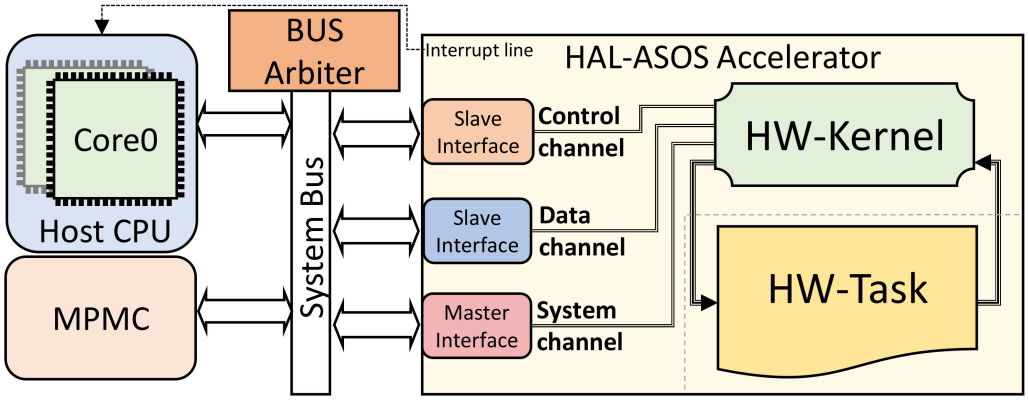

Figure 2. HAL-ASOS accelerator model integrated into host platform.

Although some works described in [1,2] provide elasticity through Dynamic Partial Reconfiguration (DPR), HAL-ASOS goes beyond the state of the art by tightly coupling such elasticity to the evolving Linux OS. Notice that today's proposed models for HW accelerators are not part of the mainstream Linux OS, because they are implemented at distinct technological levels and use specific programming languages. Future changes in the Linux OS kernel will impact synchronization, memory, or application binary interfaces (ABIs) of the proposed models, thus resulting in an increased effort to review and validate the existing designs. Using the microcode reprogrammable feature, the HAL-ASOS file system for Linux OS is extended to implement microcode page-faults at the kernel level. In doing so, the assigned exception handlers will update existing accelerators with the necessary changes to the microprogram. Furthermore, using such features will facilitate a minimal microprogram that includes just the necessary hardware system calls. Future changes on the Hardware Task design (e.g., by using DPR) with increased functionality will result in microcode faults, demanding the microprogram update to add new hardware system calls that suit the Hardware Task needs.

The remainder of this paper is organized as follows. Section 1.1 presents related work. Section 2 presents the Hardware Kernel model by describing its main building blocks, such as the Kernel Core, the hardware system calls and the microprogrammed engine, and concludes with an overview of the Linux OS integration model that supports the desired evolving elasticity. The HAL-ASOS accelerator framework is evaluated in Section 3, taking AES algorithm acceleration as a case study, and finally, Section 4 concludes the paper.

1.1. Related Work

The related work falls into four areas, as discussed in the paragraphs below: (1) Ad hoc or native FPGA acceleration; (2) operating systems for FPGA; (3) application-specific operating systems; and (4) microcode-level customization and update.

Many native FPGA-based acceleration solutions exist, which are hand-optimized for one specific application and FPGA platform, hindering productivity by demanding complete rewriting or time-consuming porting. Solutions described in [3–5] narrow their focus on data path synthesis of the hardware accelerator, completely ignoring the deep semantic integration of the hardware accelerator into the operating system, or high-level synthesis (HLS) environments as well as DPR-enabled elasticity. To reduce development time and to facilitate implementation of a complex design, HLS environments have raised the level of abstraction beyond RTL (i.e., by using high-level languages such as C/C++ or OpenCL) but following a domain-specific approach, while mixing ad hoc software and hardware abstractions, imposing obstacles to performance optimizations. Furthermore, design portability is strongly impacted when changing from one HLS environment to

another, due to their specific dependencies on custom data type, hardware support IPs, and compiler-specific "pragmas" [6].

There have been many proposals for building operating systems for FPGA, mainly due to the rise in silicon logic densities alongside the differentiating capabilities of FPGAs, such as high energy efficiency and programmability (e.g., via both static and dynamic partial reconfiguration). These features pushed FPGA from being applied as glue logic and prototyping towards implementing complete reconfigurable systems. Offloading computation to specialized hardware circuitry has been used to provide computational power and efficiency in a light-weight solution to serve the application requirements and increasing performance, while it can also be considered as complementary to complex heterogenic processor architectures. Zongwei Zhu et al. [1] propose a task scheduling framework on the DPR-based platform that exploits the differences between hardware and software tasks to improve task scheduling efficiency. ReconOS [2] extends the pioneer concept of hardware thread in FPGA as proposed by Andrews et al. [4] to a hybrid platform of CPU/FPGA, while supporting DPR for hardware threads that are scheduled through cooperative multitasking. Luca Pezzarossa et al. [5] evaluated the potential benefits of using DPR to implement hardware accelerators in real-time systems by driving the main focus towards: (1) trade-offs between hardware utilization, worst-case performance, and speed-up over a pure software solution; and (2) the trade-off between the use of multiple specialized accelerators combined with DPR instead of the use of a more general accelerator, and the memory footprint of the partial-bit streams. Hoang-Gia Vu et al. [6] propose a hardware task migration scheme assisted by (1) a checkpointing architecture for FPGAs that flattens the structure of nested modules at the HDL level, (2) a static analysis of the original HDL source code to reduce the cost of hardware, and (3) a Python-based tool to generate the checkpointing architecture at the HDL level. FOS [7] adopts a modular FPGA development flow to allow each system component to be changed and be agnostic to the heterogeneity of EDA tool versions, hardware and software layers. It dynamically maximizes the utilization transparently from the users by using resource-elastic scheduling to arbitrate the FPGA resources in both the time and spatial domain for any type of accelerators. Apples-to-apples comparison, between all these works and the HAL-ASOS accelerator model, reveals the tightly coupling elasticity to the evolving Linux OS as unique to HAL-ASOS, due to the deployed microcode-based Hardware System Call.

Several research works have been conducted on performance optimization of different features of an operating system due to the following reasons [8]: (1) OSes are critical to the performance of the running application, especially for system-intensive applications that invoke kernel features extensively, and (2) nowadays, in the cloud era, many servers only run a single application. Tarax [8] is a *one-size-fits-all* compiler-based and profile-guided optimization approach for constructing an ASOS, which modifies both the Linux OS kernel and Gnu GCC to support kernel instrumentation and profile collection. Differently from the HAL-ASOS design framework that is assisted by the mainstream and system-wide OProfile tool, Tarax does not seamlessly evolve with the Linux OS kernel as it demands both the instrumented Linux OS kernel and GCC.

Microcode is an abstraction layer between the physical components of a CPU and the programmer-visible instruction set architecture of the computer. Originally, its purpose was to simplify the design of CISC (Complex Instruction Set Computing) CPUs with the capability for in-field CPU updating without requiring any special hardware [9]. More recently, x86 microcode-level update capability has gained momentum by mitigating Spectre and Meltdown vulnerabilities. Benjamin Kollenda et al. [9] reverse engineered the microcode of x86 CPU and proposed a microcode-assisted instrumentation framework, alongside the enclave functionality, to realize a small trusted execution environment, leveraging system security defenses such as timing attack mitigations, hardware-assisted address sanitization, and instruction set randomization. CHEx86 processor architecture [10] proposes a transparent capability-based protection scheme enforced through microcode instrumentation, to defend against security exploits targeting temporal and spatial memory

safety vulnerabilities. These works are not directly compared to the evolutive elasticity of Hardware Kernel, but similar microcode mechanisms are deployed in both fields.

2. Hardware Kernel Model

The Hardware Kernel model translates the host system to the Hardware Task and provides integration at the hardware and software levels. As shown in Figure 3, it consists of a Kernel Core implementing the Control unit with a system-level Datapath, and a collection of functional units implementing the service-level Datapath. Due to space constraints, this paper only details the Hardware Kernel model at the hardware system call level. In the accelerator model, the Hardware Kernel is a static component (i.e., with the exception of the microprogramed unit contents), independent of the Hardware Task behavior, and provides resource parameters to comply with the design metrics.

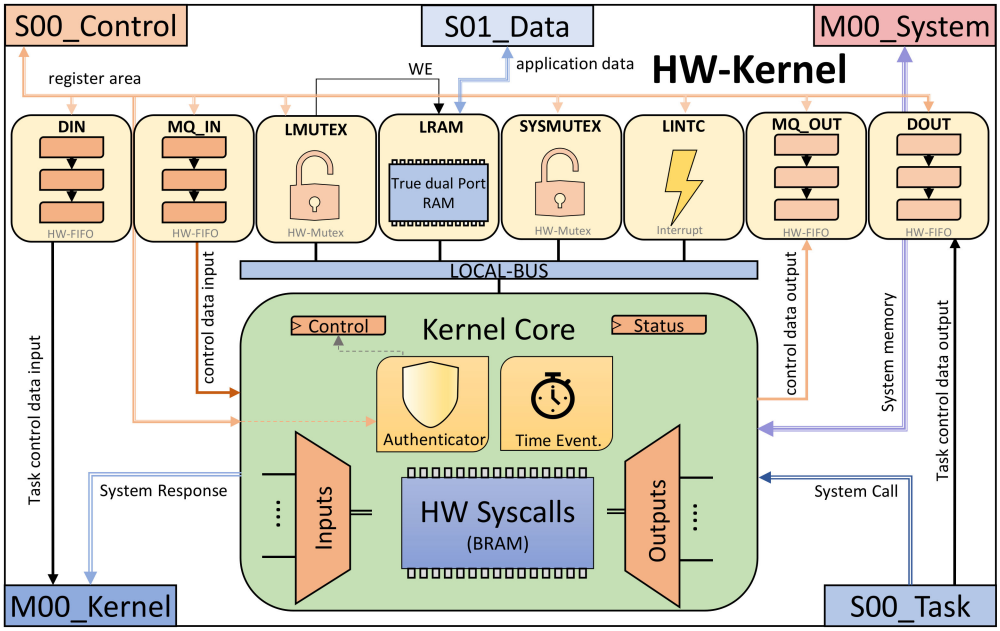

Figure 3. Hardware Kernel simplified model.

The Control unit uses a single address microcode design to encode the set of hardware system calls. The system-level Datapath implements the multiplexing and demultiplexing of the system call parameters into the service-level Datapath. The M00_Kernel and S00_Task are the master and slave interfaces of the system call, which is used to connect with the interfaces in the Hardware Task. The Kernel Core is responsible for time management and provides waiting events coupled with time-out functionalities and a parametrizable task sleep. The Control and Status registers will allow the host system to interact with the Hardware Kernel. To preserve the Hardware Kernel status, any control operation issued by the CPU (or multiple cores) is forwarded via the Authenticator unit that validates permissions before authorizing a write operation. As a consequence of the microprogramming technology used for the hardware system calls, the Kernel Core implementation results in a static unit that is independent of the Hardware Task implementation.

A service-level Datapath includes: (1) a dual-port and bidirectional message-queue used for messaging control information within the host system services; (2) a dual-port bidirectional data-FIFO available for generic Hardware Task use; (3) a local interrupt controller (*LINTC*) that allows synchronization with the Linux OS; (4) a true dual-port generic purpose local RAM (*LRAM*) for data exchange and temporary storage; and (5) two dual-channel Hardware Mutexes that implement mutual exclusion with the accelerator model. The latter are directly coupled with the LRAM and a system memory region allocated at boot time.

At the kernel side, dedicated interfaces are used to manage each of the Hardware FIFO, while the remainder of the functional units are accessed through custom Local-BUS. As depicted in Figure 3, the M00_System interface is used to access a kernel-specific region in the host system memory, the S00_Control and S01_Data offer the control- and data-oriented transfer interfaces for host system accesses to the Hardware Kernel functional units. The S01_Data implements a byte-oriented bidirectional interface used exclusively to access the LRAM. The remainder of a functional unit links to the S00_Control in a bidirectional register type interface. The complete set of units that integrate the Hardware Kernel model are also parametrizable and are made available to the host system through the Linux OS integration model.

2.1. Kernel Core

Conceptually, the Kernel Core acts like any kernel that can be found in the most elementary OS, by providing a set of services that interact with local resources through hardware system call invocation. The Hardware Task implements system calls using procedures described in a kernel Hardware Description Language (HDL) package provided by the framework. For complex or composite operations, user-level HDL procedures, provided by the user package, can implement consecutive system calls involving more than one local resource. Procedures accept input and output parameters that link to resources from the Hardware Task design. These, in turn, will allow the hardware system call to access these resources and ultimately update them with execution results.

Figure 4 shows a simplified diagram that describes the internal organization of the Kernel Core component. The Control Unit determines the *Status* of the accelerator that can be triggered by the active bits in the Control register. Such registers can be handled by the host system to address the application's functional requirements. Due to the critical nature of the available operations, the content of the Control register is updated under the supervision of the Authenticator device, which validates the received word by scanning for the required authentication field. Once active, the Control Unit operates through the system-level Datapath, establishing the connectivity between the microprogrammed unit and the kernel's Call and Response interfaces. These interfaces match directly to the S00_Task and M00_Kernel signals in the Hardware Kernel top level, and allow the Hardware Task to trigger the system calls present in the microprogrammed unit. In turn, the system calls implement a pre-programmed set of control actions, which operate at the system level, to handle adequate data manipulation using the existing local resources.

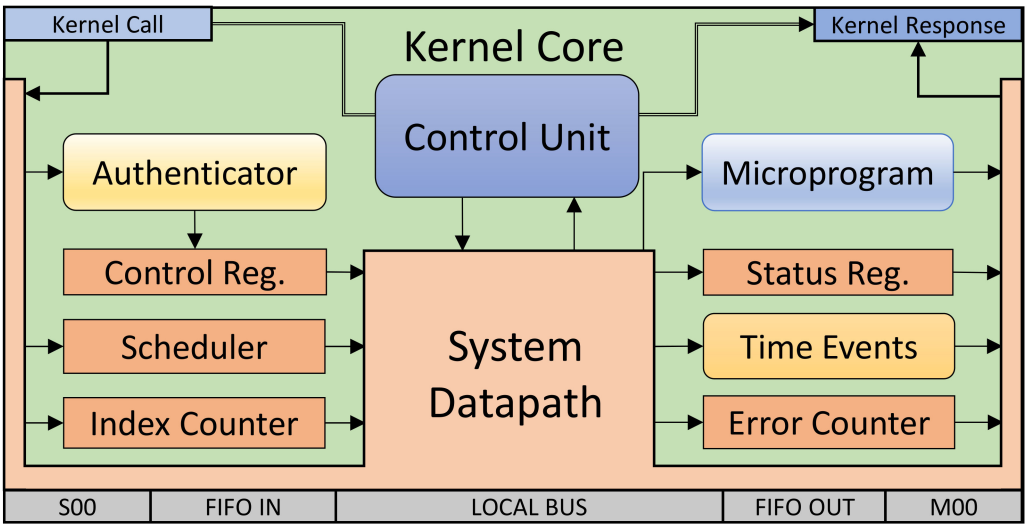

Figure 4. Kernel Core internal structure.

When a Hardware Task demands for a wait event within the duration of a predetermined number of clock cycles, or needs to wait for a hardware signal restricted to a

maximum timeout interval, a system call interacts with the Time Event device to provide such a service. In addition, to implement composite operations, the Kernel Core uses the Scheduler services to select each system call from a concurrent implementation described by a user procedure. In similar way, the Index counter is used to manipulate data using consecutive indexes. Finally, an Error counter will register any errors that may occur while executing system calls. These may lead to an error state in the Control Unit, demanding for the intervention of the host system.

Once running, the microprogrammed unit suspends the clock signal at strategic points of the Hardware Task design for all system calls. In doing so, the Hardware Task context remains suspended while it interacts at the kernel level. Pre-programmed control signals are then generated to forward the received parameters using the system-level Datapath. At the same time, status information is generated to indicate whether the system call performs a write or a read operation, or if it must stay blocked waiting for available involved resources, and also including the current microprogram location. In the final active clock cycle that completes the system call execution, the microprogram re-establishes the context on the Hardware Task, which will resume with its normal processing.

2.2. Hardware System Calls

A hardware system call is a sequence of control operations assisted by a predetermined number of steps, in order to provide services that translate local resources in the accelerator model. Similar to the concept applied at the software level in the OS environments, the hardware system calls virtualize the accelerator through a specific set of features, allowing the designer to easily create a Hardware Task. They are the Kernel Core fundamental interface, to handle the local resources and abstract away the complexity that the accelerator model represents. Such abstraction, in turn, promotes the design's reuse by allowing deployment on different platforms, as long as the set of hardware system calls offers appropriate implementation. The Kernel Core design is organized through an incremental set of programmable features.

As mentioned above, hardware system calls are implemented via procedures in the kernel package that specify the functionality, the involved parameters, and the connectivity between these and the kernel microprogram- and system-level Datapath units, while the Kernel Core provides entry and exit points in its interface that establish the required signals. Listing 1 shows an excerpt of the kernel package, defining at lines 163, 206, and 213 the *sys_call_t* type as a subset of system calls the kernel supports and uses in the input and output records to establish the system call interface. When executing system calls, each procedure specifies its arguments according to the desired feature in line 209, and links them to the input *parameters* in line 210. It then activates the *this_call* flag to signal the Kernel Core for valid inputs and to proceed with the system call. In response, the microprogram activates the *block_task* signal and transfers the received type of system call to the *syscall_id* field, line 216 and line 217, respectively. During the execution, the Kernel Core updates the *return_arg* output (line 218) with the processing results from the system-level Datapath. In the last step of the system call execution, the microprogram activates the signal on line 215, indicating valid parameters in the *return_arg* register, and at completion, it disables the *block_task* output to release the Hardware Task control. The output fields hold their contents until the next system call execution, thus allowing the Hardware Task to re-use or test them to evaluate results. Note that the kernel HDL package is inserted hierarchically, starting at the tool's configuration package. This establishes, among others, the length of the system-level Datapath determined by the largest received parameter (lines 210 and 218). This parameter is the kernel-level control message and depends on the target architecture of the host system. As a result, the length of the Datapath is fixed on two words when the tool targets a 32-bit host, or three words on a 64-bit host.

Listing 1. Kernel package source file excerpt, describing hardware system call types, entry and exit records.

```
6   library hal_asos_v4_00_a;
7   use hal_asos_v4_00_a.hal_asos_configs_pkg.all;
8   use hal_asos_v4_00_a.hal_asos_utils_pkg.all;
...
163   type sys_call_t is (SYS_CALL_NONE, SYS_CALL_WAIT_EVENT_TIMEOUT,
SYS_CALL_READ_LFIFO,
164    SYS_CALL_WRITE_LFIFO, SYS_CALL_READ_MESSAGE, SYS_CALL_WRITE_MESSAGE,
165   SYS_CALL_READ_LBUS, SYS_CALL_WRITE_LBUS, SYS_CALL_MUTEX_LOCK,
166   SYS_CALL_MUTEX_TRY_LOCK, SYS_CALL_MUTEX_UNLOCK,
SYS_CALL_READ_MBUS,
167   SYS_CALL_WRITE_MBUS, SYS_CALL_READ_LBUS_BURST,
SYS_CALL_WRITE_LBUS_BURST,
168   SYS_CALL_READ_MBUS_BURST, SYS_CALL_WRITE_MBUS_BURST, SYS_CALL_YIELD);
...
206   type sys_call_input_t is
207   record
208      this_call: std_ulogic;- - trigger sys_call
209      sys_call_id: sys_call_t;
210      parameters: std_logic_vector(C_MESSAGE_WIDTH-1 downto 0); - -field for syscall
parameters
211   end record;
212
213   type sys_call_output_t is
214   record
215      valid: std_logic;
216      block_task: std_logic;
216      sys_call_id: sys_call_t;
218      return_arg: std_logic_vector (C_MESSAGE_WIDTH-1 downto 0); - - return sys_call data
219   end record;
...
```

Algorithm 1 describes a 4-step hardware system call for the Hardware Mutex lock, where *Step0* evaluates the state of the resource and implements containment when locked. The *Locked A* flag indicates that the resource is locked by the CPU in the host platform and as such, in this particular case, the microprogram must go to *Step0* when the condition is true or proceed to *Step1*, otherwise. *Step1* acquires the resource, while *Step2* evaluates the final result of the operation. If the *Locked B* flag is set, it indicates that the resource is locked by the Kernel Core and proceeding to *Step3* releases the Hardware Task. Otherwise, the concurrent race for the resource is lost and the microprogram retries the system call invocation, returning to *Step0* until it succeeds.

Algorithm 1 Microprogram to lock a Hardware Mutex

```
1:  pseudocode SYS_CALL_MUTEX_LOCK
2:  Step0: produce block_task and lbus_rd_ce test mutex status Locked A flag.
3:         if true then goto step 0.
4:  Step1: produce block_task and lbus_wr_ce test true input.
5:         if false then goto step 2.
6:  Step2: produce block_task and lbus_rd_ce test mutex status Locked B flag.
7:         if false then goto step 0.
8:  Step3: produce valid
9:         exit
```

2.3. Microprogrammed Unit

The accelerator model employs single address microcode, and its operation is based on the flow of microinstructions of the microprogram, where each opcode activates certain

outputs and selects one input for testing. Thus, an 8-bit Program Counter advances into the next instruction according to a *true* test result, or takes a jump based on the current address and an implicit offset (*Step bit field*) in the opcode if the result is *false*. Figure 5 shows the opcode format for *Step2* of the system call to lock a Hardware Mutex. In this example, the absolute address 0×22 is applied to the RAM where the microprogram is defined. The resulting word determines that input 10 is used as a test source; "00" is the next step false (NSF), which gives rise to the absolute address 0×20 for the case of *false* test result; and output 7 remains set for the time that the current microinstruction is active. If the test result is *true*, the Program Counter is incremented to the next microinstruction, at the absolute address 0×23. It also shows the value of the outputs *Valid* (V), *Block* task (B), and *Fault* (F), which are transversal to all microinstructions, and for this reason, they are located at fixed positions in the opcode.

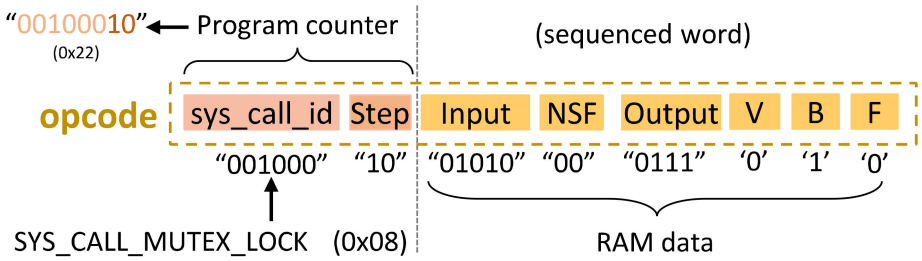

Figure 5. Microprogram-opcode format example in HW Mutex lock, Step2.

To select a test input, the design of the microprogram uses a 5-bit field (*Input*) in the opcode to implement a multiplexing function (from 32 signals to 1), which implement conditional jump, and can use "00000" or "11111" in the *Input* bit field as auxiliary *false* and *true* tests, for the unconditional jump or next instruction, respectively. In the same opcode, a 4-bit field (*Output*) allows the microprogram to activate outputs, by implementing a demultiplexer function (from 1 to 16 signals).

Table 1 shows an excerpt from the microprogram that includes the microinstructions of two system calls, the mutex lock and try-lock, while empty locations are mapped to null values for input and output with the bits *Block* and *Fault* asserted. The first signal will suspend the Hardware Task context, while the latter will trigger a Linux OS kernel page-fault. The contents in this table are ordered according to the microinstruction opcode presented in Figure 5.

Table 1. Binary excerpt from the Microprogram composed by the bit field values in the opcode, combined with the corresponding Program Counter absolute address fields.

	Sys Call ID	Step	Input	NSF	Output	Valid	Block	Fault
	...	...	...	...	...	...	...	...
SYS_CALL_MUTEX_LOCK	0x08	00	01100	00	0111	0	1	0
		01	11111	10	0110	0	1	0
		10	01010	00	0111	0	1	0
		11	00000	00	0000	1	0	0
SYS...MUTEX_TRY_LOCK	0x09	00	01100	00	0000	0	1	0
		01	11111	10	0110	0	1	0
		10	01010	00	0111	0	1	0
		11	00000	00	0000	1	0	0
	...	...	...	...	...	...	...	...
(empty)	0x30	00	00000	00	0000	0	1	1
		00	00000	00	0000	0	1	1
		00	00000	00	0000	0	1	1
		00	00000	00	0000	0	1	1
	...	...	...	...	...	...	...	...

The first line of the mutex lock system call uses the absolute address 0×20 (see example in Figure 5), where the microinstruction selects the input 12 ("01100" in the *Input* bit field) for testing a flag *Locked A*. As such, the microprogram should only proceed to the next instruction when the resource is free. In order to implement a continuous flow of valid tests, this flag must be complemented before the multiplexer input. In this way, when the *Locked A* flag is active, the input selection will result in a *false* test, and the microprogram will jump to the current instruction until the resource is released (Algorithm 1). On release, the *true* result increases the step counter, which will give rise to the next instruction in the absolute address 0×21. In this step, the microprogram activates the demultiplexer output 6 ("0110" in the Output bit field), to write in the Hardware Mutex and implements the dummy test to proceed to *Step2* on any result. For this test, it selects the auxiliary *true* logic test input statically assigned to the multiplexer input.

In the microprogram inputs, only the *Locked A* and *Locked B* are used in complemented logic, and the latter is used to test if the mutex has been released by the microprogram. As such, the same flag (without the complemented logic) is received at input 10, which gives rise to a *true* locked test. Such a test is used in *Step2* of the lock system call to ensure success in the occurrence of a race condition for the resource. Upon success, the microprogram reaches *Step3* by incrementing the step counter or otherwise, in *Step2*, a *false* test will result in conditional jump to address 0×20, and repeating the system call. In *Step3*, the output is activated to indicate valid data in the *return* register, and the Hardware Task is released by disabling the *Block* output. At completion, the microprogram needs to jump to *Step0* in the *counter* register, so that a new system call can be started. Although the increment of the counter would result in a similar behavior, the design applies a *false* test at input "00000" to favor regularity, and jumps back in the last step of each system call.

The elasticity offered by the microprogram enables services of the accelerator model to detect runtime failures, namely, failure due to unregistered addresses or wrong transaction formats while accessing the memory system. The system-level Datapath triggers the failure signal and the Kernel Core goes to a fault state, and consequently, disconnects the Hardware Task from the microprogram and asserts an interrupt signal while waiting for the file system reply. This interrupt signal triggers a Linux OS page-fault, which, when processed, checks the accelerator's *status* register and accordingly launches a specific handler for the detected failure. Each handler runs a rule-based procedure to tackle a microprogram conflict, which, for the unregistered address failure, requests Linux OS for memory allocation, followed by the forwarding of the assigned physical address to a specific purpose register of the accelerator's Hardware Kernel. Otherwise, i.e., in cases of a transaction failure, the fault is processed to identify a replacement system call that is compatible with the memory interface, and the microprogram address is reprogrammed with the newly chosen system call.

Similarly, a DPR enabling new functionalities or the replacement of a whole Hardware Task can trigger an unsupported system call by the current microprogram, since the latter only contains system calls generated during the synthesis of the original design. Thus, trying to run such a system call raises a microprogram memory fault, as any free location of the microprogram memory is mapped to null values for input and output, with the bits *Blocked* and *Fault* asserted in the sequenced word (Table 1). The Kernel Core replies accordingly by disconnecting the Hardware Task from the microprogram while going to a failure state. A rule-based procedure is selected to reprogram the accessed memory location with the required system call and triggers the Kernel Core to resume the microprogram with the newly added functionality. The S00_Control interface is used by the assigned handler to sequentially write the corresponding microprogram words, while specifying the offset of the given location. After, the handler concludes by asserting the *resume* bit in the Control register of the Kernel Core, which signals the kernel to return to a processing state and reconnect the suspended Hardware Task to the microprogram.

2.4. Linux Integration Model

The integration of the HAL-ASOS accelerator model with the Linux OS at both the user and kernel levels, and the myriad of functional units in the model, demands proper OS support for the collection of device drivers that efficiently exports each functionality into the Linux OS user-space. Such a collection is organized through a customized file system, depicted in Figure 6.

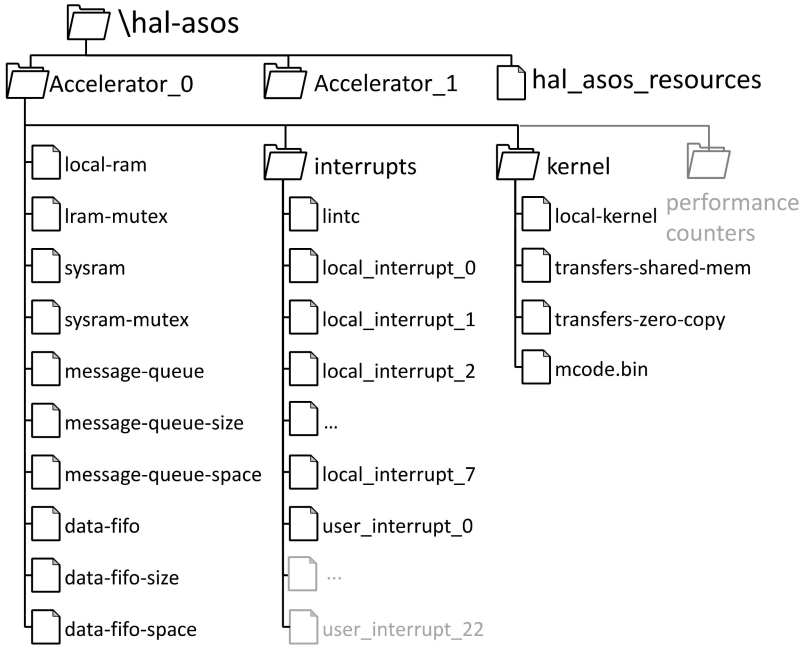

Figure 6. HAL-ASOS file system structure on Linux.

The HAL-ASOS accelerator file system is mounted at system start-up and it can be found at the root of the Linux OS file system in the *hal-asos* folder. Any existing accelerators will be probed from the device tree file and mapped into individual folders (e.g., *Accelerator_1*, ...). Inside the accelerator folder, the structure is organized into a kernel folder, an interrupt folder, and a subset of virtual files that map the remainder of functional units in the accelerator model (e.g., the local-ram, the *lram-mutex*, the *sysram* and the *sysram-mutex*, the *Hardware Kernel message-queue*, and the *data-fifo*). The interrupt folder contains the virtual files that provide the synchronization between the software threads in the system and the accelerator. The *lintc* file represents the local interrupt controller and it uses eight native interrupts and up to twenty-three user-definable interrupts, mapped to *local_** and *user_** virtual files, respectively. The kernel folder contains the *local-kernel* virtual file, used to register accelerator administrative features. Among these features, the distinct memory profile activation includes *UserIO*, *SharedMemory*, and *ZeroCopy*. The *UserIO* profile is handled at the application level through the HAL-ASOS C/C++ software framework, and the remaining profiles are implemented using the shared and zero-copy virtual files. A microcode file is used to track the changes that resulted from the accelerator faults and include these in the future system restarts.

3. AES Algorithm Acceleration: A Case Study

To evaluate the efficiency of the HAL-ASOS accelerator, an AES use case was selected among the numerous applications assessed, due to the rising demand for security in embedded applications. The ZC702 Evaluation board (Xilinx Zynq-7000 FPGA) was selected as the target platform and three distinct application versions were compared: software-only multitask; hardware-only single-task; and multitask with mixed hardware and software tasks. First, a client-side application (C programming language) that uploads files through the internet and uses the 128-bit AES algorithm to enforce security was refactored to the

HAL-ASOS framework into three *CTask* instances—a '*File reader*', an '*AES Encryptor*', and a '*Ciphered Uploader*'—communicating through the publish-subscribe Data Distribution System (DDS) provided as part of the HAL-ASOS accelerator framework. After, the refactored software-only multitask version was *OProfiled*, searching for offload candidates to be refactored as: (1) the mixed hardware and software version by offloading only the '*AES Encryptor*' to a Hardware Task and (2) the hardware-only single-task version by offloading '*File reader*', '*AES Encryptor*' and '*Ciphered Uploader*' *tasks* into only one Hardware Task. This step is assisted by the HAL-ASOS' emulator model during the mapping of pre-selected offloading candidates into the Hardware Task structure. Finally, the two designs are validated by applying the Co-Simulation model, after updating the HAL-ASOS accelerator framework library with IP-XACT description of distinct accelerator implementations.

Figure 7 shows a simplified block design for the internet application using a Vivado project targeting the ZC702 board. The functional units are implemented in the programmable logic (PL) area, among which the two hardware accelerators coupled with the two Hardware Tasks, i.e., *hw_encryptor_0* for the hardware and software version, and *hw_encryptor_1* for the hardware-only version. To deploy the accelerators in the selected hardware, the *hal_asos_accelerator_v4_00_b* component provided by the framework was used. The *v4.00.b* implements connectivity with AMBA AXI BUS using the Interconnect IPs for master and slave interfaces, and a single-clock design, which uses a 100 MHz clock frequency. Furthermore, at the application level, a console parameter is passed as an argument to select between the two accelerator implementations.

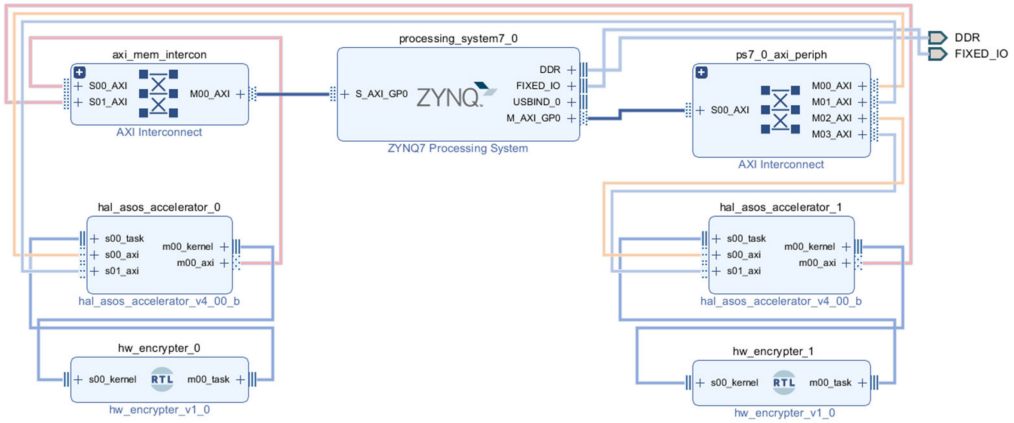

Figure 7. The simplified block design of the internet application using the ZC702 evaluation board.

Choosing an input file containing one million digits of π (i.e.,"3" + 1,000,000 digits for asymmetric input), performance results were measured by executing the '*time*' command or accessing the accelerators' performance counters. The software-only version spent 6.42 s, 8.13 s, and 1.56 s, on processing the input file (i.e., read, encrypt and upload all digits of π), running the application code in the two available CPU cores and running the Linux OS code, respectively. The mixed hardware and software multitask version spent 3.58 s for processing the input file, with 4.47 s spent in the application and 2.10 s in the Linux OS code. Lastly, the single-task hardware-only version spent 0.30 s to process the input file, with 0.13 s running the application code and 0.14 s executing the Linux OS code.

A more accurate test considers statistical data from 10 iterations on each one of the three applications. Additionally, to better characterize the application's behavior, the file length is increased by 10 and 100 times while repeating the 10 iterations. Figure 8a plots the gathered data from the total 90 applications executions; the red, the blue, and the black lines indicate the execution time of the software-only, mixed hardware and software multitask, and the single-task hardware-only versions, respectively. Please note that the execution time has been scaled almost in a linear way with the file length.

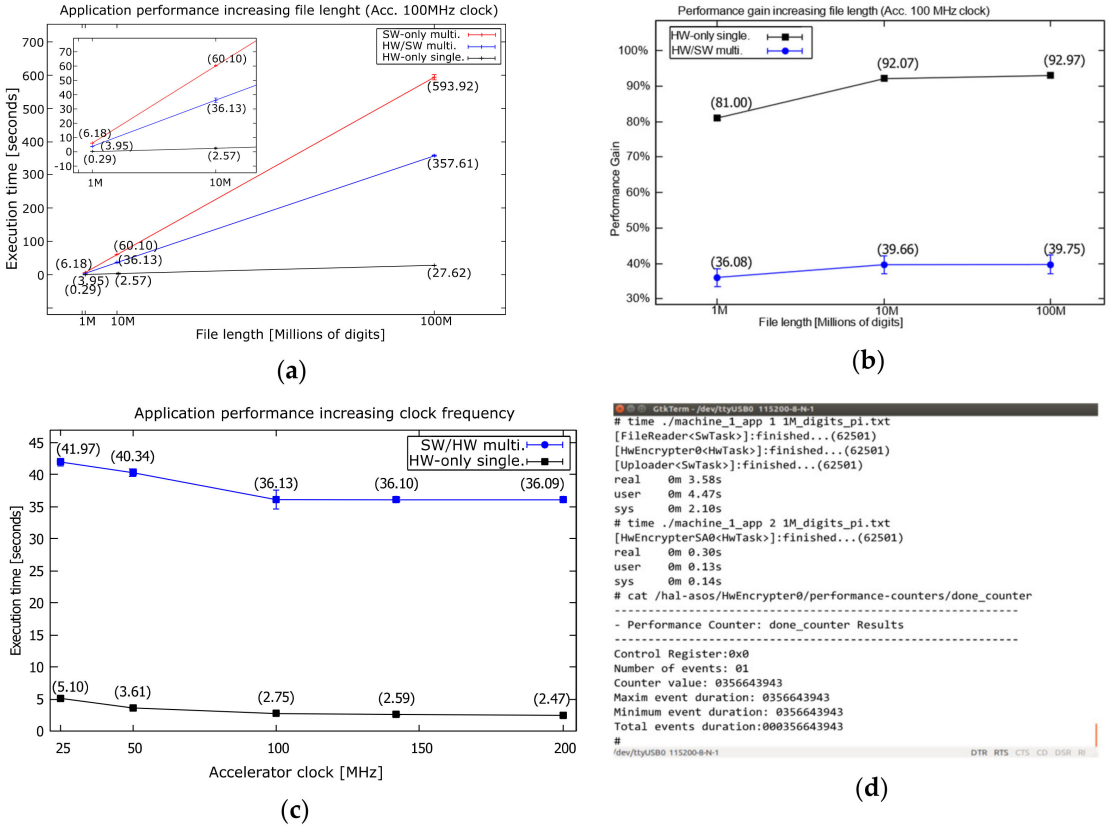

Figure 8. Performance results of: (**a**) the software-only multi-task version and the two software and hardware multi-task and single-task applications when using a 100 MHz clock frequency in the accelerators; (**b**) the two software and hardware applications over the software-only version using the same clock frequency; (**c**) the two software and hardware applications using 10 million digits while increasing the clock frequency. (**d**) The console output while executing the software and hardware applications using one-million digits of π.

The same data were plotted in a performance ratio, comparing the software-only to the hardware accelerated versions. Figure 8b depicts collected results where the blue and black lines present the performance gains achieved by offloading the mixed hardware and software multitask and the single-task hardware-only, respectively.

To better characterize the use of resources, the center value of the three input files (i.e., the 10 million digits) is chosen, while different clock sources, from 25 MHz to 200 MHz, are applied to the design with the last frequency reporting time constraint violations, but still implementable. The software-only results are not impacted by the accelerators in the PL and, for this reason, they are not considered. Figure 8c is a plot of the results gathered from 100 applications executions. The two accelerated versions increase performance by clocking the circuit until a clock frequency of 100 MHz. Beyond this value, the performance increase is imperceptible as the host system cannot respond to the increased demand of the accelerators, thus avoiding starvation of the remaining processes in the system. Figure 8d shows the console commands used to execute the two software and hardware applications at 100 MHz, using the one-million digits of π. Each task manipulates 62,501 fragments of 128-bit that correspond to the initial 1,000,002 characters and some extra padding. A performance counter from the HAL-ASOS virtual file system in the *HwEncryptor0* folder (i.e., the hardware and software multitask application) is also read to access the number of clock cycles used by the Hardware Task. Output results match with the performance results presented in Figure 8a. A total of 356,643,944 clock ticks, using a 10-nanosecond period, were used to process the one million digits file, which translates to 3.56643944 s.

4. Conclusions and Future Work

This paper extends the notion of deep semantic integration of hardware accelerators into an operating system environment proposed by *ReconOS* [2], by introducing the parametrizable Hardware Kernel unit into the HAL-ASOS accelerator model, to promote an evolutive elasticity beyond the DPR-enabled one. Such tight coupling and evolutive elasticity to Linux OS are leveraged via the microcode engine of the Hardware Kernel unit, which prevents changes on the Linux OS kernel from impacting synchronization, memory, or ABIs, as noticed on previously proposed hardware accelerator models. The HAL-ASOS file system for Linux OS was also extended to implement microcode page-faults at the kernel level, enabling easy and quick updating of Hardware System Calls.

The HAL-ASOS accelerator model is experimentally evaluated using two accelerators based on distinct Hardware Tasks that require specific Hardware System Calls to replace the software threads behavior, in handling the network subsystem and the input data using files. Results demonstrate that the concept of the Hardware System Calls provided by the microprogrammed unit suits the performance requirements of distinct AES application versions. Regarding the scalability and the processed inputted amount of data, performance gains over software-only and hardware-software implementations were also evaluated. More examples assisted by applying specific Hardware Tasks should be presented to prove the effectiveness of the proposed methodology. Additionally, more experiments must be carried out to evaluate the impact of dynamic switching and scheduling of Hardware Tasks on the DPR platform as well as accelerating Machine Learning inference engines as Hardware Tasks.

Future works will include (1) extending the proposed Hardware Kernel to provide multi-task functionality supported by specific microprogrammed engines in each dedicated Kernel Core unit, (2) evaluation of the Linux file system for the HAL-ASOS, and (3) protecting the HAL-ASOS microcode. As devices become smarter and more connected in the so-called IoT era, future work will go toward protecting the update capability of the HAL-ASOS microcode through a lightweight Hardware System Call checking, similar to the one proposed in Draco [11], by caching pre-validated Hardware Systems Calls, while keeping recently validated Hardware System Calls in a Lookaside Buffer.

Author Contributions: Conceptualization, investigation and methodology, V.S.; validation and writing—review and editing, P.P. and P.C.; resources and writing—original draft preparation J.C.; supervision, investigation and validation, A.T. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by FCT-Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zhu, Z.; Zhang, J.; Zhao, J.; Cao, J.; Zhao, D.; Jia, G.; Meng, Q. A Hardware and Software Task-Scheduling Framework Based on CPU+FPGA Heterogeneous Architecture in Edge Computing. *IEEE Open Access J.* **2019**, *7*, 148975–148988. [CrossRef]
2. Agne, A.; Happe, M.; Keller, A.; Lübbers, E.; Plattner, B.; Platzner, M.; Plessl, C. ReconOS—An Operating System Approach for Reconfigurable Computing. *IEEE Micro* **2014**, *34*, 60–71. [CrossRef]
3. Ordaz-García, O.O.; Ortiz-López, M.; Quiles-Latorre, F.J.; Arceo-Olague, J.G.; Solís-Robles, R.; Bellido-Outeiriño, F.J. DALI Bridge FPGA-Based Implementation in a Wireless Sensor Node for IoT Street Lighting Applications. *Electronics* **2020**, *9*, 1803. [CrossRef]
4. Andrews, D.; Niehaus, D.; Jidin, R.; Finley, M.; Peck, W.; Frisbie, M.; Ortiz, J.; Komp, E.; Ashenden, P. Programming models for hybrid FPGA-CPU computational components: A missing link. *IEEE Micro* **2004**, *24*, 42–53. [CrossRef]
5. Pezzarossa, L.; Kristensen, A.T.; Schoeberl, M.; Sparsø, J. Using dynamic partial reconfiguration of FPGAs in real-Time systems. *Microprocess. Microsyst.* **2018**, *61*, 198–206. [CrossRef]
6. Vu, H.G.; Nakada, T.; Nakashima, Y. Efficient hardware task migration for heterogeneous FPGA computing using HDL-based checkpointing. *Integration* **2021**, *77*, 180–192. [CrossRef]
7. Vaishnav, A.; Pham, K.D.; Powell, J.; Koch, D. FOS: A Modular FPGA Operating System for Dynamic Workloads. *ACM Trans. Reconfig. Technol. Syst.* **2020**, *13*, 1–28. [CrossRef]

8. Yuan, P.; Guo, Y.; Zhang, L.; Chen, X.; Mei, H. Building application-specific operating systems: A profile-guided approach. *Sci. China Inf. Sci.* **2018**, *61*, 092102. [CrossRef]
9. Kollenda, B.; Koppe, P.; Fyrbiak, M.; Kison, C.; Paar, C.; Holz, T. An Exploratory Analysis of Microcode as a Building Block for System Defenses. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018.
10. Sharifi, R.; Venkat, A. CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities. In Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture, Valencia, Spain, 30 May–3 June 2020.
11. Skarlatos, D.; Chen, Q.; Chen, J.; Xu, T.; Torrellas, J. Draco: Architectural and Operating System Support for System Call Security. In Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 17–21 October 2020.