OHEX: OS-Aware Hybridization Techniques for Accelerating MPSoC Full-System Simulation

Róbert Lajos Bücs*, Maximilian Fricke†, Rainer Leupers*, Gerd Ascheid*, Stephan Tobies† and Andreas Hoffmann†
* Institute for Communication Technologies and Embedded Systems, RWTH Aachen University, Germany
{*buecs, leupers, ascheid*}*@ice.rwth-aachen.de*
† Synopsys GmbH, Aachen, Germany {*Maximilian.Fricke, Stephan.Tobies, Andreas.Hoffmann*}*@synopsys.com*

***Abstract*—Virtual platform (VP) technology is an established enabler of embedded system design. However, the sheer number of CPU models in modern multi-core VPs forms a performance bottleneck. Hybrid simulation addresses this issue by executing parts of the embedded software stack on the host. Although the approach is significantly faster, hybridization can not cope with higher software layers, e.g., Operating Systems (OSs). Thus, this paper presents the OS-aware Host EXtension (OHEX) framework to accelerate VPs while expanding the applicability of hybridization. OHEX is evaluated on various system layers, yielding speedups between 2.99x-21.14x with specific benchmarks.**

***Index Terms*—virtual platforms, hybridization, OS-awareness**

I. Introduction

The application of embedded systems has increased drastically over the past decade. Fast deployment of new technologies is in great demand across various industry verticals, e.g., internet of things and automotive, among others. As a result, the complexity of *software* (SW) applications is exploding, entailing the utilization of powerful *Multi Processor System on Chip* (MPSoC) *hardware* (HW) devices. However, the design, test, verification and validation steps of such full systems have become immensely difficult due to their sheer complexity.

Virtual Platforms (VPs): This technology has become an established enabler of embedded system design, tackling the previously mentioned challenges. VPs consist of simulation models of HW blocks, created via *Electronic System-Level* (ESL) standards, e.g., *SystemC/TLM* [1]. Such standards enable modeling and simulation of diverse components, e.g., *Instruction Set Simulators* (ISSs), interconnects, memories, various peripheral devices, as well as full systems. Herein the embedded SW stack can also be simulated as part of the VP, enabling HW/SW co-design. Moreover, VPs facilitate system prototyping by providing full HW/SW visibility, debuggability and non-intrusive monitoring, while also ensuring simulation determinism. However, the sheer number of ISS models to be simulated in modern multi-core VPs forms a performance bottleneck, greatly limiting execution efficiency.

Native Execution: One way to improve simulation performance is raising the abstraction level. For this, parts of the embedded SW are compiled for and executed on the native host, which is generally orders of magnitude faster then ISS-based simulation. Yet, only a limited subset of the embedded SW can be compiled for the host, excluding architecture-specifics (e.g., drivers, inline assembly). Lastly, native execution lacks even approximate time annotation, restricting its usage for ESL.

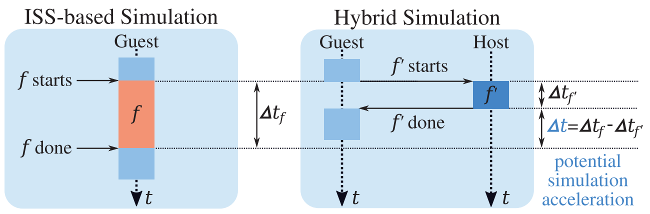


Fig. 1: Motivation: potential speedup of hybrid simulation.

Hybrid Simulation [2]: Joining the former ideas, this technique proposes to dynamically switch between an ISS-based VP (*guest*) and native execution (*host*), while performing state synchronization. This approach leverages both high simulation coverage of the guest and vast execution efficiency of the host. A simplified example for hybrid simulation is shown in Fig. 1. Herein the original guest SW block f is replaced by its native substitute f' executed on the host via special instrumentation. Since f' is completed inherently faster, the total execution time is reduced by $\Delta t = \Delta t_f - \Delta t_{f'}$ accelerating the simulation. Although hybridization is an established technique to speed up VPs, limitations arise with higher guest SW layers, e.g., full-fledged *Operating Systems* (OSs) that, to the best knowledge of the authors, have not yet been thoroughly examined.

Contributions: This paper presents OS-aware hybridization techniques to accelerate VP-based MPSoC full-system simulation. For this, guest SW blocks are to be substituted with individually developed native replacement snippets, named *Host Extensions* (HEs). Ideally, any part of the guest SW can be targeted hereby, from single functions to OS layers ultimately. Putting this in practice, the *OS-aware Host EXtension* (OHEX) framework is presented, providing fundamental hybridization mechanisms, as well as *Application Programming Interfaces* (APIs) for HE creation. OHEX is evaluated via several HE prototypes targeting various system layers (e.g., user-space applications, functions in the target OS kernel), highlighting its capability to cover a wide spectrum of use-cases.

II. The OHEX Hybrid Simulation Techniques

OHEX implements essential mechanisms and provides APIs for hybrid simulation. Shown in Fig. 2a, HE development is based on these, wrapping around manually developed native substitutes of guest SW blocks, instrumenting them for hybridization. The tools, techniques and fundamental facilities OHEX is based on, as shown in the figure, are detailed next.

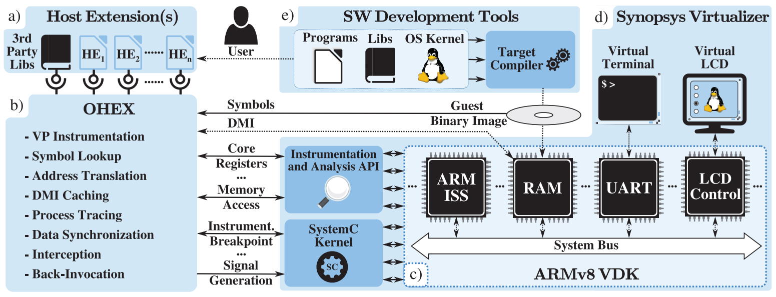
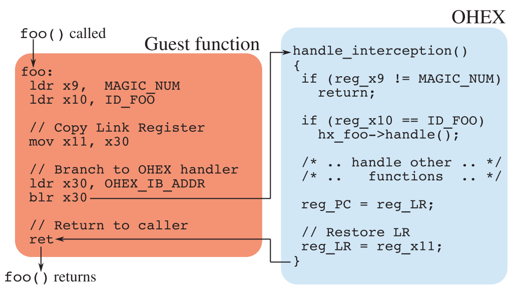


Fig. 2: Full hybrid simulation framework composition.



```
foo:
 ldr x9,  MAGIC_NUM
 ldr x10, ID_FOO

 // Copy Link Register
 mov x11, x30

 // Branch to OHEX handler
 ldr x30, OHEX_IB_ADDR
 blr x30

 // Return to caller
 ret
```

```
handle_interception()
{
 if (reg_x9 != MAGIC_NUM)
   return;

 if (reg_x10 == ID_FOO)
   hx_foo->handle();

 /* .. handle other .. */
 /* ..   functions  .. */

 reg_PC = reg_LR;

 // Restore LR
 reg_LR = reg_x11;
}
```

Fig. 3: Interception trigger meta-code example.

A. *VP Instrumentation*

OHEX builds upon the Synopsys Virtualizer [3] simulation environment (Fig. 2d). Beyond system introspection, the tool provides special VP Instrumentation and Analysis APIs [4], granting external applications control over the simulation. Shown in Fig. 2b-c, OHEX uses these to, e.g., get/set ISS registers and access memory, needed for higher level mechanisms. Herein symbol lookup is also supported to, e.g., locate functions and global data structures in the guest image. For memory inspection, the instrumentation API uses by default *SystemC/TLM transfer* packages which can cause a simulation bottleneck with frequent reads/writes. To improve this, OHEX extends the Virtualizer API by using the SystemC *Direct Memory Interface* (DMI) [1], granting immediate pointer access to guest memory locations. This feature was further enhanced by DMI caching to accelerate repeated transfers. The API also offers *Instrumentation Breakpoints* (IBs) to suspend the simulation if the ISS reaches certain addresses. For this, IBs provide function callbacks to external applications which OHEX exploits to intercept guest SW blocks and to invoke individual HEs. This separates OHEX from, e.g., semi-hosting approaches, which either require to modify the simulator or utilize special ISS instructions (e.g., trap) for interception.

B. *Target System Awareness*

HW Insight: OHEX was used with an off-the-shelf VP, called *ARMv8 Virtualizer Development Kit* (VDK) [5], shown in Fig. 2c. This contains an ARM *big.LITTLE* [6] ISS, including two quad-core CPU clusters, a system bus, memory, various peripherals (e.g., UART, LCD controller) and virtual devices (e.g., terminal, LCD) sufficient to run full-fledged OSs. OHEX incorporates device-specific know-how to support HE design for close-to-hardware layers (e.g., drivers). This often requires to influence the system state, e.g., by triggering *interrupts* (IRQs). More importantly, a deep insight into the CPU architecture is essential, e.g, instruction set, register layout and privilege levels, among many other characteristics.

Program/System State: CPU properties and the procedure call standard establish the system programming model, including the *program state* (i.e., general-purpose registers and certain memory segments). These attributes sufficiently define the state of programs written purely in C, as they follow well-established calling conventions. Yet, guest SW containing assembly code might bear side-effects on the global *system state*. In case of the used ARMv8 [7] ISS, e.g., the *supervisor monitor call* (`SVC`) instruction generates an exception. As such actions are challenging to be reproduced by HEs, they either need to be abstracted away or fall back to the ISS.

OS-awareness: As shown in Fig. 2e, the VDK executes a full-fledged embedded GNU/Linux. Herewith HE design becomes challenging as Linux separates addresses into kernel and user space via virtual memory management. Therefore, OHEX supports *(i)* guest virtual to physical address translation and *(ii)* guest physical to host virtual address mapping. Yet, since user-space processes share the same virtual address ranges, *memory aliasing* still arises, i.e., foreign processes may trigger IBs due to matching addresses. Herein symbol lookup is not enough, as certain executables and shared libraries are dynamically loaded to addresses determined at runtime. For this, OHEX monitors process creation and context switches within the kernel and fetches program names and unique identifiers from OS-internal task descriptors. For shared libraries, OHEX may trace the dynamic linker to detect which executables link against a target library. Although this technique is non-invasive to the simulation, it would possibly introduce an overhead which can be avoided, as detailed next.

C. *Fundamental Hybridization Mechanisms*

Based on the previously discussed techniques, the fundamental mechanisms of hybrid simulation can be implemented:

Interception: The process of bypassing a guest SW block and executing its HE on the host. As a faster alternative to process/linker tracing, OHEX uses *interception triggers* with user-space applications. Shown in Fig. 3, target guest blocks are instrumented to activate interceptions. First, a magic number, a unique function ID (`ID_FOO`) and the return address (from the link register `x30`) are passed via caller-saved registers (`x9`-`x11`) [7]. OHEX uses a single IB on a fixed address (`OHEX_IB_ADDR`) to fetch interceptions. On hit, it identifies the triggering block (via `ID_FOO`) and invokes its HE. After execution, the program counter is set to the original return address and the simulation resumes with the ISS jumping to the caller function. Yet, interception triggers require to modify and recompile the target SW. To avoid intrusion, future work addresses injecting interception triggers into the guest binary.

Synchronization: Required to keep the guest functionally consistent by capturing its state when an interception occurs and restoring it before resuming the simulation. To achieve this, HEs also need to convert utilized data structures between

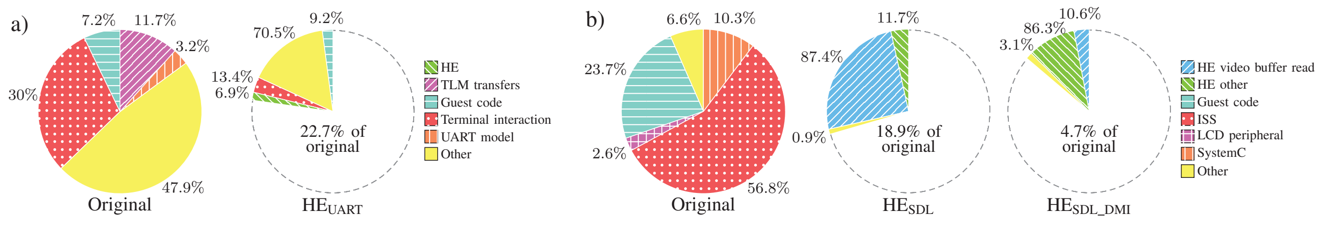


Fig. 4: Application execution ratios of the UART driver (a) and the video rendering layer (b) without and with HEs.

their guest and host representations, which might differ in terms of endianness, type sizes and memory layout.

Back-Invocation: The process of calling a guest function from within the context of an HE, required when a replacement can not or should not be provided for it, and needs execution on the ISS (e.g., due to side effects). To achieve this, OHEX emulates a guest function call following the calling convention. By invoking the target block, the simulation is resumed. After the function returns, OHEX fetches the return value, restores the state and re-activates the suspended HE. Yet, back-invocation also suffers from memory aliasing. Thus, a generic *back-invocation wrapper* was added to the guest SW that forwards calls to the function(s) to be invoked. Lastly, further efforts will be invested in injecting the back-invocation wrapper directly into the guest binary to avoid recompilation.

III. Evaluation of OHEX via HE Prototypes

This section presents HE prototypes built with OHEX, aiming to speed up the execution of embedded HW/SW layers.

A. UART Prototype

The first module, called HE$_{\text{UART}}$, speeds up console interactions over the UART of the VDK. This HE is especially relevant, as sending data over the UART was found performance-critical with frequent calls. When programs request transmission, the data is first buffered by a mediator layer (`tty`). Once enough gathered, the UART driver is called to start the transmission. The driver unmasks the transmission IRQ and, on acknowledgment, copies the input character-wise into the UART data register, which sends it to the virtual terminal.

HE$_{\text{UART}}$ first intercepts the call of the mediator for starting the transmission. Instead of writing single characters to the UART data register, it reads the whole buffer from guest memory. It then passes the input directly to the virtual terminal and clears the buffer in guest memory, as if the driver operated normally. HE$_{\text{UART}}$ bypasses parts of the driver and the UART model, thus reducing TLM transfers (detailed in Sec. II-A).

The prototype was evaluated via a user-space program sending 130kB of data to the terminal. Fig. 4a shows profiling results, depicting relative execution times of various parts of the simulation. `Guest code` denotes the benchmark process, the mediator and the driver, from which the latter was reduced in HE$_{\text{UART}}$ by mostly bypassing the CPU. `Other` covers the ISS execution engine and related SystemC activations. Utilization of the `UART model` is fully removed, as HE$_{\text{UART}}$ sends data directly to the virtual terminal, denoted as `Terminal interaction`. The activity of this block is reduced as well by passing more coarse-grained data chunks. Related `TLM transfers` were also erased, as HE$_{\text{UART}}$ does not use TLM transactions to access guest memory. Lastly, the `HE` activity itself contains interception and buffer read/write operations.

Shown in Fig. 4a, HE$_{\text{UART}}$ reduced the overall benchmark runtime to 22.7% of the original execution, corresponding to a **speedup of 4.4x**, as indicated by the scaled size of the chart.

B. SDL Video Rendering Prototype

The next HE prototypes target to accelerate video rendering. During video playback on embedded Linux, the media player utilizes *FFmpeg* [8] first to read frames from the disk, decode and convert them into a common format, and then pass them to the *Simple Directmedia Layer* (SDL) [9] for scaling and rendering. Lastly, SDL uses the framebuffer via an OS driver to display the video on the virtual LCD device of the VDK.

The prototypes, named HE$_{\text{SDL}}$ / HE$_{\text{SDL\_DMI}}$, speed up the performance-critical scaling and rendering in SDL by bypassing the guest library with its host equivalent. For this, the same version of SDL was compiled for guest and host. HE$_{\text{SDL}}$ instruments the guest variant by replacing functions with interception triggers. After frame pre-processing, the SDL rendering routine is invoked by the guest. As SDL is a user-space library, OS-aware mechanisms were employed to intercept this call. SDL routines also often allocate memory for internal data structures that HE$_{\text{SDL}}$ replicates on the host and synchronizes at guest-host transitions. After reading a frame from the guest, synchronization, scaling and rendering occurs on the host. Lastly, HE$_{\text{SDL\_DMI}}$ extends HE$_{\text{SDL}}$ by accelerating guest video buffer accesses via DMI (as detailed in Sec.II-A).

Fig. 4b shows profiling results during video playback. Compared to the original run, the HEs almost wholly bypass SDL-related `Guest code` and thus the `ISS` execution engine. The `LCD peripheral` is fully unused as the host SDL utilizes the host graphics libraries directly. The reduced `SystemC` and `Other` component activations arise due to the side effect of limited time annotation in OHEX. This causes the rendering of numerous video frames to occur in a single quantum, leading to fewer synchronizations between the SystemC thread of the ISS and the rest of the simulation. As such side effects are undesirable, as future work, timing annotation via profiling execution times will be explored. Lastly, HE$_{\text{SDL\_DMI}}$ further reduces `HE Video buffer read` time via DMI.

Shown in Fig. 4b, HE$_{\text{SDL}}$ reduced the original benchmark execution time to 18.9% while HE$_{\text{SDL\_DMI}}$ to even 4.7%. These correspond to significant **speedups of 5.29x** and **21.14x** respectively, yet with a notable loss in simulation accuracy.

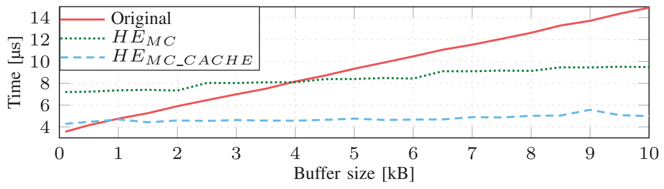


Fig. 5: Time of 10^4 `memcpy()` calls with various buffer sizes.

C. Memory Copy Prototype

The prototypes HE_{MC} / $\text{HE}_{\text{MC\_CACHE}}$ follow up on accelerating memory transfers. Memory copy is realized by the Linux kernel `memcpy()` routine, written as hand-optimized ARM inline assembly. HE_{MC} intercepts the guest `memcpy()` and fetches the virtual addresses and sizes of the source / destination buffers via function arguments. After locating the buffers in host memory, HE_{MC} calls the `memcpy()` of the host C standard library, and uses DMI for data transfer. As mapping between virtual guest and host address spaces is not necessarily linear, fragmentation of guest buffers is possible in host memory in which case `memcpy()` falls back to the guest. In other cases, to avoid virtual to physical address translation at every copy, $\text{HE}_{\text{MC\_CACHE}}$ applies additional host pointer caching, beneficial for repeated copies (discussed in Sec. II-A).

The HEs were evaluated via a user program invoking 10^4 `memcpy()` calls with varying buffers sizes. Fig. 5 shows copy times during benchmark execution. Herein the performance of the HEs heavily depends on the buffer sizes. Although for small buffers the guest execution is faster, HE_{MC} outperforms the original in case of buffers larger than 4kB. $\text{HE}_{\text{MC\_CACHE}}$ narrows this gap further, making its usage beneficial from 1kB buffers, achieving **speedups** up to **2.99x** with 10kB buffers.

IV. Related Work

Various approaches emerged to speed up VPs via hybridization. Krämer et al. present HySim [2], a tool switching between guest and host at function boundaries. HySim includes simulation control, as well as ISS and memory synchronization. The tool also incorporates an automated instrumenter, compiling architecture-independent guest functions natively. Although this is beneficial for productivity, authors acknowledge constraints with non-virtualizable functions, e.g., state altering routines, function pointers, direct stack manipulation and assembly. With missing back-invocation and OS-awareness, HySim is inoperable for higher level SW layers.

In [10], authors present a hybridization method using guest binary manipulation. Alike OHEX, the HEs are to be provided manually. Herein the semi-hosting API [11] of the used ARM ISS was extended to integrate interception and back-invocation directly into the model, that generally requires vendor support. The complexity of the proposed binary replacement is revealed, as authors achieve interception by injecting a host branch instruction into the ISS execution engine, and save/restore its state via a newly introduced ISS mode.

A further approach extends the guest generic graphics accelerator [12] to speed up ARM Mali [13] GPU simulations by using the host GPU. First, an interception layer was added to the guest OS redirecting original graphic library calls. Next, the used ARM ISS was extended by a plugin enabling guest-host communication. When the interception layer fetches a call, a message is sent to the ISS. This invokes a graphic library emulator on the host, executing the equivalent API function and returning the result. Although this idea is similar to the HEs presented in Sec. III-B, the mechanism requires vendor support for altering the ISS, strongly limiting its applicability.

In contrast to previous works, OHEX is not bound to architecture-independent SW and allows switching beyond function boundaries. For simulation inspection and control, OHEX relies only on VP instrumentation APIs. Although this may add certain overhead, OHEX is independent of the used ISS and does not need vendor support. OHEX works with individually developed HEs. Although their design requires a deep understanding of the target SW, this allows for arbitrary substitutes, enabling to exploit architectural traits of the host. Lastly, OHEX adds OS-awareness to intercept user-space applications, that none of the presented approaches accomplish.

V. Conclusion

This paper presented ISS-agnostic, OS-aware hybridization techniques for accelerating full-system simulators. To put this in practice, OHEX was presented, providing fundamental mechanisms for HE development. For evaluation, several HE prototypes were implemented, highlighting the feasibility of the framework in a wide spectrum of use cases, yielding speedups between 2.99x-21.14x with specific benchmarks.

As future work, further HE design is planned, e.g., for Linux kernel decompression. To improve time annotation, means for profiling and estimation will be explored. Lastly, to facilitate HE development, the combination of OHEX and frameworks such as [2] is planned, providing developers HE templates to perform automatic state synchronization, while utilizing the presented less intrusive instrumentation mechanisms.

References

[1] "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp. 1–638, Jan 2012.

[2] S. Krämer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr, "Hysim: A fast simulation framework for embedded software development," in *Conference on HW/SW Codesign and System Synthesis (CODES+ISSS)*, Sept 2007, pp. 75–80.

[3] "Synopsys Virtualizer - Official Product Website," https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html, Accessed Aug. 2017.

[4] *Virtual Prototype Software Analysis User's Guide*, Synopsys Inc., 2017.

[5] *ARMv8 Base VDK for ARM Cortex Processors User Guide*, Synopsys, Inc., September 2015, available at https://solvnet.synopsys.com/.

[6] ARM big.LITTLE Technology, ARM Holdings plc, https://developer.arm.com/technologies/big-little, Accessed Aug. 2017.

[7] *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*, ARM Holdings plc, September 2016, available at http://arm.com.

[8] "FFmpeg Home Page," https://ffmpeg.org, Accessed Aug. 2017.

[9] "Simple DirectMedia Layer," https://www.libsdl.org, Accessed Aug. 2017.

[10] J. Lee, D. Paterson, S. O'Neill, H. Kim, S. Nam, and J. Kim, "Binary replacement technique for application programming interface level simulation," in *Proceedings of Embedded and Real Time Software and Systems (ERTS2)*, 2010.

[11] *Angel Debug Protocol Messages*, ARM Holdings plc, September 1999, Online available: http://infocenter.arm.com/help/topic/com.arm.doc.dui0053d/ADP_ARM_DUI0053D.pdf. Accessed Aug. 2017.

[12] *Generic Graphics Accelerator User Guide*, ARM Holdings plc, May 2016, Online available: http://infocenter.arm.com/help/topic/com.arm.doc.100534_0100_01_en/generic_graphic_accelerator_ug_100534_0100_01_en.pdf. Accessed Aug. 2017.

[13] ARM Mali graphics processor. Online available: http://www.arm.com/products/graphics-and-multimedia/mali-gpu. ARM Holdings plc. Accessed Aug. 2017.