Theory of Multi Core Hypervisor Verification

Ernie Cohen[1], Wolfgang Paul[2], and Sabine Schmaltz[2]

[1] Microsoft
ecohen(at)microsoft.com
[2] Saarland University, Germany
{wjp,sabine}(at)wjpserver.cs.uni-saarland.de

Abstract. From 2007 to 2010, researchers from Microsoft and the Verisoft XT project verified code from Hyper-V, a multi-core x-64 hypervisor, using VCC, a verifier for concurrent C code. However, there is a significant gap between code verification of a kernel (such as a hypervisor) and a proof of correctness of a real system running the code. When the project ended in 2010, crucial and tricky portions of the hypervisor product were formally verified, but one was far from having an overall theory of multi core hypervisor correctness even on paper. For example, the kernel code itself has to set up low-level facilities such as its call stack and virtual memory map, and must continue to use memory in a way that justifies the memory model assumed by the compiler and verifier, even though these assumptions are not directly guaranteed by the hardware. Over the last two years, much of the needed theory justifying the approach has been worked out. We survey progress on this theory and identify the work that is left to be done.

1 Introduction and Overview

Low-level system software is an important target for formal verification; it represents a relatively small codebase that is widely used, of critical importance, and hard to get right. There have been a number of verification projects targetting such code, particularly *operating system* (OS) kernels. However, they are typically designed as providing a proof of concept, rather than a viable industrial process suitable for realistic code running on modern hardware. One of the goals of the Verisoft XT project [1] was to deal with these issues. Its verification target, the hypervisor Hyper-V [2] was highly optimized, concurrent, shipping C/assembler code running on the most popular PC hardware platform (x64). The verification was done using VCC, a verifier for concurrent C code based on a methodology designed to maximize programmer productivity – instead of using a deep embedding of the language into a proof-checking tool where one can talk directly about the execution of the particular program on the particular hardware.

We were aware that taking this high-level view meant that we were creating a non-trivial gap between the abstractions we used in the software verification and the system on which the software was to execute. For example,

– VCC has an extension allowing it to verify x64 assembly code; why is its approach sound? For example, it would be unsound for the verifier to assume that hardware registers do not change when executing non-assembly code, even though they are not directly modified by the intervening C code.

– Concurrent C code (and to a lesser extent, the C compiler) tacitly assumes a strong memory model. What justifies executing it on a piece of hardware that provides only weak memory?
– The hypervisor has to manage threads (which involves setting up stacks and implementing thread switch) and memory (which includes managing its own page tables). But most of this management is done with C code, and the C runtime already assumes that this management is done correctly (to make memory behave like memory and threads behave like threads). Is this reasoning circular?

When we started the project, we had ideas of how to justify all of these pretenses, but had not worked out the details. Our purpose here is to i) outline the supporting theory, ii) review those parts of the theory that have already been worked out over the last few years, and iii) identify the parts of the theory that still have to be worked out.

1.1 Correctness of Operating System Kernels and Hypervisors

Hypervisors are, at their core, OS kernels, and every basic class about theoretical computer science presents something extremely close to the correctness proof of a kernel, namely the simulation of k one-tape *Turing machines* (TMs) by a single k-tape TM [3]. Turning that construction into a simulation of k one-tape TMs by a single one-tape TM (*virtualization* of k *guest* machines by one *host* machine) is a simple exercise. The standard solution is illustrated in figure 1. The tape of the host machine is subdivided into tracks, each representing the tape of one of the guest machines (*address translation*). Head position and state of the guest machines are stored on a dedicated field of the track of that machine (a kind of *process control block*). Steps of the guests are simulated by the host in a round robin way (a special way of *scheduling*). If we add an extra track for the data structures of the host and add some basic mechanisms for communications between guests (*inter process communication*) via *system calls*, we have nothing less than a one-tape TM kernel. Generalizing from TMs to an arbitrary computation model M (and adding I/O-devices), one can specify an M kernel as a program running on a machine of type M that provides

– virtualization: the simulation of k guest machines of type M on a single host machine of type M
– system calls: some basic communication mechanisms between guests, I/O devices, and the kernel

At least as far as the virtualization part is concerned, a kernel correctness theorem is essentially like the Turing machine simulation theorem, and can likewise be conveniently expressed as a forward simulation. For more realistic kernels, instead of TMs we have processors, described in dauntingly large manuals, like those for the MIPS32 [4] (336 pages), PowerPC [5] (640 pages), x86 or x64 [6, 7] (approx. 1500, resp. 3000 pages). The TM tape is replaced by RAM, and the tape head is replaced by a *memory management unit* (MMU), with address translation driven by in-memory *page tables*. Observe that a mathematical model of such machine is part of the definition of correctness for a 'real' kernel.

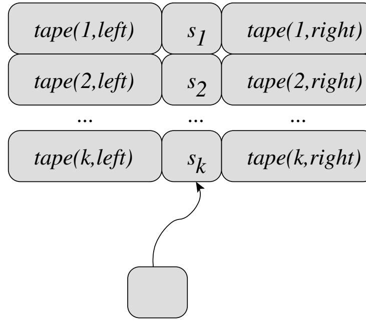

Fig. 1. Simulating k Turing machines with 1 k-band Turing machine.

Hypervisors are kernels whose guests are themselves operating systems or kernels, i.e. each guest can run several *user processes*. In terms of the TM simulation, each guest track is subdivided into subtracks for each user, each subtrack having its own process control block; the actual tape address for the next operation of a process can be calculated from its tape address, the layout of the subtrack within the track, and the layout of the track itself. In a real kernel, the address of a memory access is calculated from a virtual address using two levels of address translation, the first level of traslation provided by the guest to users via the *guest page tables* (GPTs), and the second provided by the hypervisor to the guest. On many recent processors, this second level of address translation is provided in hardware by a separate set of *host page tables*. On processors providing only a single level of translation, it is possible to take advantage of the fact that the composition of two translations is again a translation, and so can be provided by a single set of page tables. Because these *shadow page tables* (SPTs) correspond to neither the guest nor the host page tables, they are constructed on the fly by the hypervisor from the GPTs, and the hypervisor must hide from the guest that translation goes through these tables rather than the GPTs. Thus, the combined efforts of the hypervisor and the MMU simulate a virtual MMU for each guest.

1.2 Overview

We discuss the following seven theories in the remainder of the paper:

Multi-Core ISA-sp We define a nondeterministic concurrent *instruction set architecture* (ISA) model, suitable for system programming. In addition to processor cores and main memory, it includes low-level (but architecturally visible) features such as store buffers, caches, and memory management units. Ideally, this would be given in (or at least derived from) the system programmer's manuals published by the chip manufacturers. In reality, many subtle (but essential) details are omitted from these manuals. Indeed, hardware manufacturers often deliberately avoid commiting themselves to architectural boundaries, to maximize their flexibility in optimizing the implementation, and many details leveraged by real operating systems (such as details concerning the walking of page tables) are shared only with their most

important customers, under agreements of nondisclosure. Such fuzzy architectural boundaries are acceptable when clients are writing operating systems; a progammer can choose whether to program to a conservative model (e.g., by flushing translations after every change to the page tables) or program more aggressively to a model that takes advantage of architectural details of the current processor generation. But such a fuzzy model is fatal when trying to build an efficient hypervisor, because the architectural specification must both be strong enough for client operating systems to run correctly, yet weak enough that it can be implemented efficiently on top of the available hardware.

We provide evidence for the particular model we use and for the particular ways in which we resolved ambiguities of the manuals in the following way: i) we define a simplified ISA-sp that we call MIPS-86, which is simply MIPS processor cores extended with x86-64 like architecture features (in particular, memory system and interrupt controllers), ii) we reverse engineer the machine in a plausibly efficient way at the gate level, and iii) we prove that the construction meets the ISA-sp, and iv) we confirm with OS engineers that the model is sufficiently strong to support the memory management algorithms used in real operating systems. The correctness theorems in this theory deal with the correctness of hardware for multi-core processors at the gate level.

ISA Abstraction Multi-core machines are primarily optimized to efficiently run ordinary user code (as defined in the user programming manuals). In this simplified instruction set (ISA-u), architectural details like caches, page tables, MMUs, and store buffers should be transparent, and multithreaded programs should see sequentially consistent memory (assuming that they follow a suitable synchronization discipline). A naive discipline combines lock-protected data with shared variables, where writes to shared variables flush the store buffer. A slightly more sophisticated and efficient discipline requires a flush only when switching from writing to reading [8]. After proper configuration, a simulation between ISA-sp and ISA-u has to be shown in this theory for programs obeying such disciplines.

Serial Language Stack A realistic kernel is mostly written in a high-level language (typically C or C++) with small parts written in macro assembler (which likewise provides the stack abstraction) and even smaller parts written in plain assembler (where the implementation of the stack using hardware registers is exposed, to allow operations like thread switch). The main definition of this theory is the formal semantics of this computational model. The main theorem is a combined correctness proof of optimizing compiler + macro assembler for this mixed language. Note that compilers translate from a source language to a clean assembly language, i.e. to ISA-u.

Adding Devices Formal models for at least two types of devices must be defined: regular devices and interrupt controllers (the APIC in x86/64). A particularly useful example device is a hard disk – which is needed for booting. Interrupt controllers are needed to handle both external interrupts and interrupt-driven interprocess communication (and must be virtualized by the hypervisor since they belong to the architecture). Note that interrupt controllers are very particular kinds of devices in the sense that they are interconnected among each other and with processor cores in a way regular devices are not: They inject interrupts collected from regular de-

vices and other interrupt controllers directly into the processor core. Thus, interrupt controllers must be considered specifically as part of an ISA-sp model with instantiable devices[3]. Crucial definitions in this theory are i) sequential models for the devices, ii) concurrent models for ISA-sp with devices, and iii) models for single core processors semantics of C with devices (accessed through *memory mapped I/O* (MMIO)). The crucial theorems of this theory show the correctness of drivers at the code level.

Extending the Serial Language Stack with Devices to Multi-Core Machines The crucial definition of this theory is the semantics of concurrent 'C + macro assembly + ISA-sp + devices'. Besides ISA-sp, this is *the* crucial definition of the overall theory, because it defines the language/computational model in which multi-core hypervisors are coded. Without this semantics, complete code level verification of a hypervisor is not meaningful. Essentially, the ownership discipline of the ISA abstraction theory is lifted to the C level; in order to enable the implementation of the ownership discipline, one has to extend serial C with volatile variables and a small number of compiler intrinsics (fences and atomic instructions). In this theory there are two types of major theorems. The first is compiler correctness: if the functions of a concurrent C program obeying the ownership discipline are compiled separately, then the resulting ISA-u code obeys the ownership discipline and the multi-core ISA-u code simulates the parallel C code. The second is a reduction theorem that allows us to pretend that a concurrent C program has scheduler boundaries only just before actions that race with other threads (I/O operations and accesses to volatile variables).

Soundness of VCC and its Use Programs in the concurrent C are verified using VCC. In order to argue that the formal proofs obtained in this way are meaningful, one has to prove the soundness of VCC for reasoning about concurrent C programs, and one has to show how to use VCC in a sound way to argue about programs in the richer models.

Obviously, for the first task, syntax and semantics of the annotation language of VCC has to be defined. VCC annotations consist essentially of "ghost" (a.k.a. "auxilliary" or "specification") state, ghost code (used to facilitate reasoning about the program, but not seen by the compiler) and annotations of the form "this is true here" (e.g. function pre/post-conditions, loop invariants, and data invariants). Then three kinds of results have to be proven. First, we must show that if a program (together with its ghost code) is certified by VCC, then the "this is true here" assertions do in fact hold for all executions. Second, we must show that the program with the ghost code simulates the program without the ghost code (which depends on VCC checking that there is no flow from ghost state to concrete state, and that all ghost code terminates). Third, we must show that the verification implies that the program conforms to the Cohen/Schirmer [8] ownership discipline (to justify VCC's assumption of a sequentially consistent model of concurrent C).

To reason about richer programming models with VCC, we take advantage of the fact that the needed extensions can be encoded using C. In particular, one can add

[3] MIPS-86 provides such an ISA-sp model with interrupt controllers and instantiable devices – albeit currently at a level where caches are already invisible.

additional ghost data representing the states of processor registers, MMUs and devices to a C program; this state must be stored in "hybrid" memory that exists outside of the usual C address space but from which information can flow to C memory. We then represent assembly instructions as function calls, and represent active entities like MMUs and devices by concurrently running C threads.

Hypervisor Correctness The previous theories serve to provide a firm foundation for the real verification work, and to extend classical verification technology for serial programs to the rich computational models that are necessarily involved in (full) multi-core hypervisor verification. Verification of the hypervisor code itself involves several major components, including i) the implemention of a large numbers of 'C + macro assembly + assembly' threads on a multi-core processor with a fixed small number of cores, ii) for host hardware whose MMUs do not support two levels of translations, the correctness of a parallel shadow page table algorithm, iii) a TM-type simulation theorem showing virtualization of ISA-sp guests by the host, and iv) correct implementation of system calls.

2 ISA Specification and Processor Correctness

2.1 Related Work

For single core RISC (*reduced instruction set computer*) processors, it is well understood how to specify an ISA and how to formally prove hardware correctness. In the academic world, the papers [9] and [10] report the specification and formal verification of a MIPS-like processor with a pipelined core with forwarding and hardware interlock, internal interrupts, caches, a fully IEEE compliant pipelined floating point unit, a Tomasulo scheduler for out of order execution, and MMUs for single-level pages tables. In industry, the processor core of a high-end controller has been formally verified [11]. To our knowledge, there is no complete formal model for any modern commercial CISC (*complex instruction set computer*); until recently, the best approximations to such a model were C simulators for large portions of the instruction set [12–14].

The classical memory model for multi-core processors is Lamport's sequentially consistent shared memory [15]. However, most modern multi-core processors provide efficient implementations only of weaker memory models. The most accurate model of the memory system of modern x86/64 architectures, "x86-tso", is presented in [16]. This model abstracts away caches and the memory modes specifying the cache coherence protocol to be used, and presents the memory system as a sequentially consistent shared memory, with a separate FIFO store buffer for each processor core. It is easy to show that the model collapses if one mixes in the same computation cacheable and non cacheable memory modes on the same address (accesses in non cacheable memory modes bypass the cache; accesses in different non cacheable modes have different side effects on the caches). That the view of a sequentially consistent shared memory can be maintained even if of one mixes in the same computation accesses to the same address with different "compatible" memory modes/coherence protocols is claimed in the classical paper introducing the MOESI protocol [17], but we are not aware of any proof of this fact.

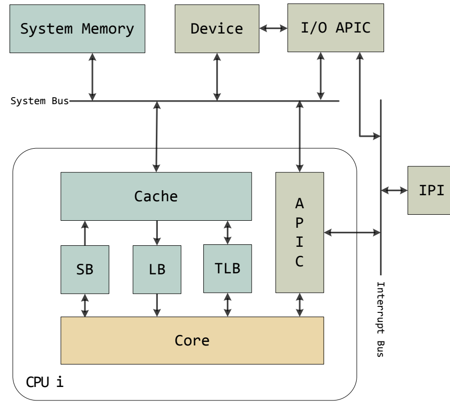

Fig. 2. x86-64 processor model consisting of components whose steps are interleaved nondeterministically.

Another surprising observation concerns correctness proofs for cache coherence protocols. The model checking literature abounds with papers showing that certain invariants of a cache system are maintained in an interleaving model where the individual cache steps are atomic. The most important of these invariants states that data for the same memory address present in two caches are identical and thus guarantees a consistent view on the memory at all caches. For a survey, see [18]. These results obviously provide an important step towards the provably correct construction of a sequentially consistent shared memory. Apart from our own results in [19], we are not aware of a gate-level construction of hardware main memory, caches, cache controllers, and the busses connecting them for which it has been proved that parallel execution of hardware accesses to the caches simulates the high-level cache model.

2.2 Modeling an x86-64-like ISA-sp

A formal model of a very large subset of the x64 ISA-sp was constructed as part of the Hyper-V verification project, and is presented in [20]. This 300 page model specifies 140 general purpose and system programming instructions. Due to time constraints, the model omits debug facilities, the alignment check exception, virtual-8086 mode, virtual interrupts, hardware task-switching, system management mode, and devices other than the local APICs. The MMX extension of the instruction set is formalized in the complementary thesis [21]. The instruction set architecture is modeled by a set of communicating nondeterministic components as illustrated in figure 2. For each processor, there is a processor core, MMU, store buffer, caches (which become visible when accesses of non cacheable and cacheable memory modes to the same address are mixed in the same computation), and a local APIC for interrupt handling. The remaining components (shared between the cores) are main memory and other devices. Sizes of caches, buffers, and translation look aside buffers (TLBs) in the MMU are unbounded in the model, but the model is sufficiently nondeterministic to be implemented by an implementation using arbitrary specific sizes for each of these. In the same spirit, caches

and MMUs nondeterministically load data within wide limits, allowing the model to be implemented using a variety of prefetch strategies. Nevertheless, the model is precise enough to permit proofs of program correctness.

As mentioned in the introduction, an accurate ISA-specification is more complex than meets the eye. Only if the executed code obeys a nontrivial set of software conditions, the hardware interprets instructions in the way specified in the manuals. In RISC machines, the alignment of accesses is a typical such condition. In pipelined machines, the effects of certain instructions only become visible at the ISA level after a certain number of instructions have been executed, or after an explicit pipeline flush. In the same spirit, a write to a page table becomes visible at the ISA level when the instruction has left the memory stages of the pipe, the write has left the store buffer, *and* previous translations effected by this write are flushed from the TLB by an INVLPG instruction (which in turn does only become visible when it has left the pipe). In a multi-core machine, things are even more complicated because a processor can change code and page tables of other processors. In the end, one also needs *some* specification of what the hardware does if the software violates the conditions, since the kernel generally cannot exclude their violation in guest code. In turn, one needs to guarantee that guest code violating software conditions does not violate the integrity of other user processes or the kernel itself. Each of these conditions exposes to the ISA programmer details of the hardware, in particular of the pipeline, in a limited way.

Obviously, if one wants to verify ISA programs, one has to check that they satisfy the software conditions. This raises the problem of how to identify a *complete* set of these conditions. In order to construct this set, we propose to reverse engineer the processor hardware, prove that it interprets the instructions set, and collect the software conditions we use in the correctness proof of the hardware. Reverse engineering a CISC machine as specified in [20] is an extremely large project, but if we replace the CISC core by a MIPS core and restrict memory modes to 'write back' (WB) and 'uncacheable' (UC) (for device accesses), reverse engineering becomes feasible. A definition of the corresponding instruction set called 'MIPS-86' fits on 44 pages and can be found in [22].

2.3 Gate Level Correctness for Multi-Core Processors

A detailed correctness proof of a multi-core processor for an important subset of the MIPS-86 instruction set mentioned above can be found in the lecture notes [19]. The processor cores have classical 5 stage pipelines, the memory system supports memory accesses by bytes, half words, and words, and the caches implement the MOESI protocol. Caches are connected to each other by an open collector bus and to main memory (realized by dynamic RAM) by a tri-state bus. There are no store buffers or MMUs, yet. Caches support only the 'write back' mode. The lecture notes contain a gate-level correctness proof for a sequentially consistent shared memory on 60 pages. Integrating the pipelined processor cores into this memory system is not completely trivial, and proving that this implements the MIPS-86 ISA takes another 50 pages. The present proof assumes the absence of self-modifying code.

Dealing with tri-state busses, open collector busses, and dynamic RAM involves design rules, which can be formulated but not motivated in a gate-level model. In analogy

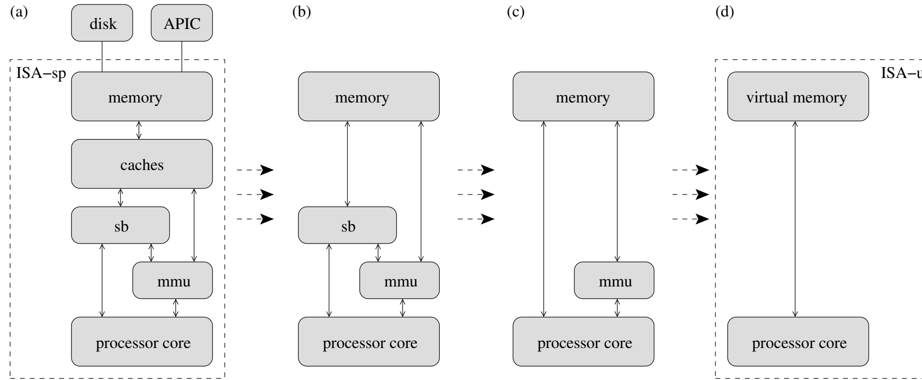

Fig. 3. Abstracting ISA-sp to ISA-u.

to data sheets of hardware components, [19] therefore uses a detailed hardware model with minimal and maximal propagation delays, enable and disable times of drivers, and setup and hold times of registers as its basic model. This allows derivation of the design rules mentioned above. The usual digital hardware model is then derived as an abstraction of the detailed model.

2.4 Future Work

The correctness proof from [19] has to be extended in the following ways to cover MIPS-86

- introducing a read-modify-write operation (easy),
- introducing memory fences (easy),
- extending memory modes to include an uncacheable mode UC (easy),
- extending the ISA-sp of MIPS-86 with more memory modes and providing an implementation with a coherence protocol that keeps the view of a single memory abstraction if only cacheable modes are used (easy)
- implementing interrupt handling and devices (subtle),
- implementing an MMU to perform address translation (subtle),
- adding store buffers (easy), and
- including a Tomasulo scheduler for out-of-order execution (hard).

3 Abstracting ISA-sp to ISA-u

One abstracts ISA-sp to ISA-u in three steps: i) eliminating the caches, ii) eliminating the store buffers, and iii) eliminating the MMUs. A complete reduction (for a naive store buffer elimination discipline and a simplified ownership discipline) is given in [23].

3.1 Caches

To eliminate caches, an easy simulation shows that the system with caches S simulates a system without caches S'; the coupling invariant is that for every address, the S' value at that address is defined as the value cached at that address if it is cached (the value is unique, since all cache copies agree), and is the value stored in the S memory if the location is uncached. One gets the processor view of figure 3 (b).

3.2 Store Buffers and Ownership Disciplines

In single-core architectures, an easy simulation shows that the system with FIFO store buffers S simulates a system without store buffers S': the value stored at an address in S' is the value in the store buffer farthest away from memory (i.e., the last value saved) if the address appears in the store buffer, and is otherwise the value stored in the memory of S. For a single-core architecture, no extra software conditions are needed; a more careful proof can be found in [24]. One gets the view of figure 3 (c). For the multi-core architecture, store buffers can only be made invisible if the executed code follows additional restrictions.

A trivial (but highly impractical) discipline is to use only flushing writes (which includes atomic read-modify-write operations); this has the effect of keeping the store buffers empty, thus rendering them invisible. A slightly more sophisticated discipline is to classify each address as either shared or owned by a single processor. Unshared locations can be accessed only by code in the owning processor; writes to shared addresses must flush the store buffer. The proof that this simulates a system without store buffers is almost the same as in the uniprocessor case: for each owned address, its value in the S' memory is the value in its owning processor according to the uniprocessor simulation, and for each shared address, the value is the value stored in memory.

A still more sophisticated discipline to use the same rule, but to require a flush only between a shared write and a subsequent share read on the same processor. In this case, a simple simulation via a coupling invariant is not possible, because the system, while sequentially consistent, is not linearizable. Instead, S' issues a write when the corresponding write in S actually emerges from the store buffer and hits the memory. S' issues a shared read at the same time as S; this is consistent with the writes because shared reads happen only when there are no shared writes in the store buffer. The unshared reads and writes are moved to fit with the shared writes[4]. It is straightforward to extend this reduction to include shared "read-only" memory.

A final, rather surprising improvement to the last reduction discipline is to allow locations to change from one type to another programatically. For example, we would like to have a shared location representing a lock, where an ordinary operation on that lock (acquiring it) gives the thread performing that action ownership of some location

[4] Note that this means that in S', an unshared write from one processor might be reordered to happen before a shared write from another processor, even though the shared write hits memory first, so while the execution is sequentially consistent, it is not "memory sequential consistent" as defined in [25], because it violates the triangle race condition. Allowing sequentially consistent executions with triangle races is an absolute requirement for practical reduction theorems for x86-TSO.

protected by that lock. Moreover, we might want to allow the set of locations protected by that lock to change, perhaps determined by data values. [8] gives a very general reduction theorem for x86-TSO that allows these things to be done in the most flexible way possible, by allowing the program to take ownership of shared data, give up ownership of data, and change it between being read-only and read-write, in ordinary ghost code. This theorem says that if you can prove, *assuming sequential consistency*, that a concurrent program (which includes ghost code that might change memory types of locations) follows (a minor modification of) the flushing discipline above, then the program remains sequentially consistent when executed under x86-TSO. The proof of this reduction theorem is much more difficult than the previous ones, because reducing a single TSO history requires reasoning from the absence of certain races in related sequentially consistent histories.

3.3 Eliminating MMUs

Modern processors use page tables to control the mapping of virtual to physical addresses. However, page tables provide this translation only indirectly; the hardware has to walk these page tables, caching the walks (and even partial walks) in the hardware TLBs. x86/64 machines require the system programmer to manage the coherence of the TLBs in response to changes in the page tables. The simplest way to make MMUs invisible is to set up a page table tree that represents an injective translation (and does not map the page tables themselves), before switching on virtual memory. It is an easy folklore theorem that the resulting system simulates unvirtualized memory; a proof can be found in [24]. One gets the view of figure 3 (d). However, this is not how real kernels manage memory; memory is constantly being mapped in and unmapped. The easiest way to do this is to map the page tables in at their physical addresses (since page table entries are based on physical, rather than virtual, page frame numbers). At the other extreme, one can model the TLBs explicitly, and keep track of those addresses that are guaranteed to be mapped to a particular address in all possible complete TLB walks (and to not have any walks that result in a page fault), and to keep track of a subset of these addresses, the "valid" addresses[5], such that the induced map on these addresses is injective. Only those addresses satisfying these criteria can be read or written. This flexibility is necessary for kernels that manage memory agressively, trying to minimize the number of TLB flushes. Essentially, this amounts to treating the TLB in the same way as a device, but with the additional proof obligation connecting memory management to reading and writing, through address validity. This, however, we currently consider future work.

3.4 Mixed ISA-sp and ISA-u Computations

In a typical kernel, there is a stark contrast between the kernel code and the user programs running under the kernel. The kernel program needs a richer model that includes system instructions not accessible to user programs, but at the same time the kernel can

[5] Note that validity of an address is, in general, different for different processors in a given state, since they flush their TLB entries independently.

be written using a programming discipline that eliminates many irrelevant and mathematically inconvenient details. For example, if the kernel is being proved memory-safe, the programming model in the kernel does not have to assign semantics to dereferencing of null pointers or overrunning array bounds, whereas the kernel must provide to user code a more complex semantics that takes such possibilities into account. Similarly, the kernel might obey a synchronization discipline that guarantees sequential consistency, but since user code cannot be constrained to follow such a discipline, the kernel must expose to user code the now architecturally-visible store buffers. In the case of a hypervisor, the guests are themselves operating systems, so the MMU, which is conveniently hidden from the hypervisor code (other than boot code and the memory manager), is exposed to guests.

3.5 Future Work

Extension of the work in [23] to a full a proof of the naive store buffer reduction theorem should not be hard. In order to obtain the reduction theorem with dirty bits, it is clearly necessary to extend the store buffer reduction theorem of [8] to machines with MMUs. This extension is not completely trivial as MMUs directly access the caches without store buffers. Moreover MMUs do not only perform read accesses; they write to the 'accessed' and 'dirty' bits of page table entries. One way to treat MMUs and store buffers in a unified way is to treat the TLB as shared data (in a separate address space) and the MMU as a separate thread (with an always-empty store buffer). This does not quite work with the store buffer reduction theorem above; because the TLB is shared data, reading the TLB to obtain an address translation for memory access (which is done by the program thread) would have to flush the store buffer if it might contain a shared write, which is not what we want. However, the reduction theorem of [8] can be generalized so that a shared read does not require a flush as long as the same read can succeed when the read "emerges" from the store buffer; this condition is easily satisfied by the TLB, because a TLB of unbounded capacity can be assumed to grow monotonically between store buffer flushes.

4 Serial Language Stack

4.1 Using Consistency Relations to Switch Between Languages

As explained in the introduction, realistic kernel code consists mostly of high-level language code, with some assembler and possibly some macro assembler. Thus, complete verification requires semantics for programs composed of several languages L_k with $0 \leq k < n \in \mathbb{N}$. Since all these languages are, at some point, compiled to some machine code language L, we establish for each L_k that programs $p \in L_k$ are translated to programs $q \in L$ in such a way that *computations* (d^i) – i.e. sequences of configurations $d^i, i \in \mathbb{N}$ – of program q simulate computations (c^i) of program p via a consistency relation $consis(c, d)$ between high level configurations c and low level configurations d. Translation is done by compilers and macro assemblers. Translators can be optimizing or not. For non-optimizing translators, steps of language L_k are translated into one

or more steps of language L. One shows that, for each computation (c^i) in the source language, there is a step function s such that one has

$$\forall i : consis(c^i, d^{s(i)})$$

If the translator is optimizing, consistency holds only at a subset of so called 'consistency points' of the source language. The translator does not optimize over these points. Let $CP(i)$ be a predicate indicating that configuration c^i is a consistency point. Then an optimizing translator satisfies

$$\forall i : CP(i) \rightarrow consis(c^i, d^{s(i)})$$

Note that the consistency relation and the consistency points together specify the compiler. The basic idea to formulate mixed language semantics is very simple. We explain it here only for two language levels (which can be thought of as machine code and high level abstract semantics, as in figure 4); extension to more levels is straightforward and occurs naturally when there is a model stack of intermediate languages for compilation[6]. Imagine the computations (c^i) of the source program and (q^j) of the target program as running in parallel from consistency point to consistency point. We assume the translator does not optimize over changes of language levels, so configurations c^i where the language level changes are consistency points of the high level language. Now there are two cases

- switching from L_k to L in configuration c^i of the high level language: we know $consis(c^i, d^{s(i)})$ and continue from $d^{s(i)}$ using the semantics of language L.
- switching from L to L_k in configuration d^j of the low level language: we try to find a configuration c' of the high level language such that $consis(c', d^j)$ holds. If we find it we continue from c' using the semantics of the high level langue. If we do not find a consistent high level language configuration, the low level portion of the program has messed up the simulation and the semantics of the mixed program switches to an error state.

In many cases, switching between high-level languages L_k and L_l by going from L_k to shared language L, and from the resulting configuration in L to L_l can be simplified to a direct transition from a configuration of L_k to a configuration of L_l by formalizing just the compiler calling convention and then proving that the resulting step is equivalent to applying the two involved consistency relations (e.g., see [26]). This explains why the specification of compilers necessarily enters into the verification of modern kernels.

4.2 Related Work

The formal verification of a non-optimizing compiler for the language C0, a type safe PASCAL-like subset of C, is reported in [27]. The formal verification of an optimizing compiler for the intermediate language C-minor is reported in [28]. Mixed language

[6] Note in particular, that, when two given high level language compilers have an intermediate language in common, we only need to switch downwards to the highest level shared intermediate language.

semantics for C0 + in line assembly is described in [29] as part of the Verisoft project [11]. Semantics for C0 + external assembly functions is described in [30]. Both mixed language semantics were used in sebsequent verification work of the Verisoft project. As the underlying C0 compiler was not optimizing, there was only a trivial calling convention. Note that the nontrivial calling conventions of optimizing compilers produce proof goals for external assembly functions: one has to show that the calling conventions are obeyed by these functions. Only if these proof goals are discharged, one can show that the combined C program with the external functions is compiled correctly.

4.3 A Serial Language Stack for Hypervisor Verification

Similar to [28], we use an intermediate language C-IL with address arithmetic and function pointers. The semantics of C-IL *together* with a macro assembler obeying the same calling conventions is described in [26]. Calls from C-IL to macro assembly and vice versa are allowed. To specify the combined semantics, one has to describe the ABI (i.e. the layout of stack frames and the calling convention used). In [31], an optimizing compiler for C-IL is specified, a macro assembler is constructed and proven correct, and it is shown how to combine C-IL compiler + macro assembler to a translator for combined C-IL + macro assembly programs. As explained in the introduction, extension of this language stack to C-IL + macro assembly + assembly is necessary to argue about saving and restoring the base and stack pointers during a process switch or a task switch. This can be done using the construction explained in subsection 4.1.

4.4 Future Work

We believe that, for the consistency relations normally used for specifying compilers and macro assemblers, the mixed language semantics defined in subsection 4.1 is essentially deterministic in the following sense: if $consis(c, d)$ holds, then d is unique up to portions of c which will not affect the future I/O behavior of the program (e.g. non reachable portions of the heap). A proof of such a theorem should be written down.

5 Adding Devices

The obvious way to add devices is to represent them as concurrent threads, and to reason about the combined program in the usual way. This approach is justified only if the operational semantics of the language stack executing the program in parallel with the device models simulates the behavior of the compiled code running on the hardware in parallel with the devices. This is already nontrivial, but is further complicated by the addition of interrupts and interrupt handlers. It is obviously undesirable to introduce interrupt handling as a separate linguistic concept, so the natural way to model an interrupt handler is as a concurrent thread. However, the relationship between a program and an interrupting routine is somewhat closer than that between independent threads; for example, data that might be considered "thread local" in the context of a concurrent program might nevertheless be modified by an interrupt handler, which requires careful management of when interrupts are enabled and disabled. Another complication that arises with many kinds of devices is the need to capture and model real-time constraints.

5.1 Related Work

Formal methods have been extremely successful identifying large classes of frequent bugs in drivers [32]. In contrast, *complete* formal verification results of even of the most simple drivers have only recently appeared. An obvious prerequisite is a formal model of devices that can be integrated with processor models both at the hardware and ISA level [33]. At the hardware level, processor and devices work in parallel in the same clock domain. At the hardware level, the models of some devices are completely deterministic; an example is a dedicated device producing timer interrupts. But these models also can have nondeterministic portions, e.g. the response time (measured in hardware cycles) of a disk access. When we lift the hardware construction to the ISA model, one arrives in a natural way at a nondeterministic concurrent model of computation: processor and device steps are interleaved in an order not known to the programmer at the ISA level or above. This order observed at the ISA level can be constructed from the hardware construction and the nondeterminism stemming from the device models. A formal proof for the correctness of such a concurrent ISA model for a single core 'processor + devices' was given in [34, 35]. The hardware construction for catching the external interrupts and the corresponding correctness argument are somewhat tricky due to an - at first sight completely harmless - nonstandard specification in the instruction set of the underlying processor, which was taken from [36]. There, external interrupts are defined to be of type 'continue', i.e. the interrupted instruction is completed before the interrupt is handled. In the MIPS-86 instruction set [22] mentioned above, this definition was changed to reflect standard specification, where external interrupts are of type 'repeat', i.e. the interrupted instruction is not executed immediately, but is instead repeated after the run of the handler.

Now consider a system consisting of a (single core) processor and k devices as shown in figure 5, and consider a run of a driver for device i. Then one wants to specify the behavior of the driver by pre and post conditions. For example, if the driver writes a page from the processor to the disk, the precondition would state that the page is at a certain place in processor memory and the post condition would specify that it is stored at a certain place on the memory of the disk. To prove this one has to show that the other devices do not interfere with the driver run. Indeed one can show an order reduction theorem showing that if during the driver run i) interrupts of other devices are disabled and ii) the processor does not poll the devices, then in a driver run with arbitrary interleaving all steps of devices $\neq i$ can be reordered such that they occur after the driver run without affecting the result of the computation. A formal proof of this result is given in [37, 30]. At the same place and in [38] the integration of devices into the serial model stack of the Verisoft project (resulting in C0 + assembly + devices) and the formal verification of disk drivers is reported.

Note that the above reorder theorem for device steps has the nontrivial hypothesis that there are no side channels via the environment, i.e. the outside world between the devices. This is not expliciteley stated; instead it it is implicitly assumed by formalizing figure 5 in the obvious way. For example, if device 2 is a timer triggering a gun aimed at device 1 during the driver run of device 1, the post condition is false after the run because the device is not there any more. Side channels abound of course, in particular

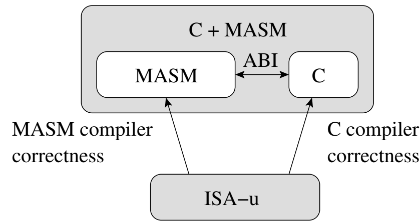

Fig. 4. Combined semantics of C and Macro Assembler.

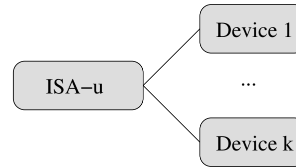

Fig. 5. ISA model with devices.

in real time systems. If device 1 is the motor control and device 2 the climate control in a car, then the devices *are* coupled in the environment via the power consumption.

5.2 Multi-Core Processors and Devices

Only some very first steps have been made towards justifying verification of multi-core processors along with devices. The current MIPS-86 instruction set contains a generic device model and a specification of a simplified APIC system consisting of an I/O APIC (a device shared between processors that distributes device interrupts) and local APICs (processor local devices for handling external and inter processor interrupts). Rudimentary models of interrupt controllers for various architectures have been built as parts of VCC verifications.

5.3 Future Work

Instantiating the generic device model of MIPS-86 with an existing formal disk model is straightforward. Justification of the concurrent 'multi-core processor + devices model' of the MIPS-86 ISA requires of course the following steps

- extending the MIPS-ISA hardware from [19] with the pipelined interrupt mechanism from [10]. The catching and triggering of external interrupts needs to be modified to reflect that external interrupts are now of type 'repeat'. This should lead to a simplification of the construction.
- reverse engineering of hardware APICs and the mechanism for delivering inter processor interrupts (IPIs).
- showing that the hardware constructed in this ways interprets the MIPS-86 ISA. This proof should be simpler than the proof in [34].

Three more tasks remain open: i) proving a reorder theorem for driver runs, ii) the reduction theorem from multi-core ISA-sp to ISA-u has to be generalized to the situation, where the hardware contains devices, and iii) devices must be integrated in the serial model stack (resulting in C-IL + macro assembly + assembly + devices) along the lines of [37, 30, 38]. These results would justify language-level reasoning about device drivers in multi-core systems.

Formally verifying secure booting is another interesting research direction: 'secure boot' guarantees that the verified hypervisor (if we have it) is indeed loaded and run. This involves the use of verified hardware and libraries for crypto algorithms. Such libraries have already formally been verified in the Verisoft project [11].

6 Extending the Serial Language Stack to Multi-Core Computations

Language stacks deal with languages at various levels and translations between them. A 'basic' language stack for multi-core computations consists simply of i) a specification of some version of 'structured parallel C', ii) a compiler from this version of parallel C to the ISA of a multi-core machine and iii) a correctness proof showing simulation between the source program and the target program. Definitions of versions of structured parallel C (or intermediate languages close to it) and correctnes proofs for their compilers proceed in the flowing way:

- one starts with a small-step semantics of the serial version of the high level language; configurations of such semantics have program rests/continuations, stacks, and global memory. Configurations for parallel C are easily defined: keep program rests and stack local for each thread; share the global variables among threads. Computations for unstructured parallel C are equally easy to define: interleave steps of the small steps semantics of the individual threads in an arbitrary order.
- compiling unstructured parallel C to multi-core machines tends to produce very inefficient code. Thus one structures the computation by restricting accesses to memory with an ownership discipline very similar to the one of [8]. Different versions of parallel C differ essentially by the ownership discipline used. As a directive for the compiler, variables which are allowed to be unowned and shared (such that they can e.g. be used for locks) are declared as volatile. Accesses to volatile variables constitute *I/O-points* of the computation.
- Compilers do not optimize over I/O-points, thus I/O-points are consistency points. Except for accesses to volatile variables, threads are simply compiled by serial compilers. Code produced for volatile accesses has two portions: i) possibly a fence instruction draining the local store buffer; clearly this is only necessary if the target machine has store buffers, and ii) an appropriate atomic ISA instruction.
- Compiler correctness is then argued in the following steps: i) The compiled code obeys the ownership discipline of the target language in such a way that volatile accesses of the compiled code correspond to volatile accesses of the source code, i.e. I/O points are preserved. Then one proves both for the source language and the target language that, due to the ownership discipline, memory accesses between I/O points are to local, owned, or shared-read-only addresses only. This implies at both language levels an order reduction theorem restricting interleavings to occur at I/O points only. We call such an interleaving an *I/O-block schedule*. iii) One concludes simulation between source code and target code using the fact that I/O points are compiler consistency points and thus in each thread compiler consistency is maintained by the serial (!) computations between I/O-points.

6.1 Related Work

The 'verified software toolchain' project [39] presently deals with a 'basis' language stack. C minor is used as serial source language. The serial compiler is the formally verified optimizing compiler from the CompCert project [28]. Permissions on memory are modified by operations on locks – this can be seen as a kind of ownersip discipline. The target machine has sequentially consistent shared memory in the present work; draining store buffers is identified as an issue for future work. Proofs are formalized in Coq. In the proofs the permission status of variables is maintained in the annotation language. We will return to this project in the next section.

6.2 Extending the Language Stack

A crucial result for the extension of a language stack for 'C + macro assembly + ISA-sp + devices' to the multi-core world is a general order reduction theorem that allows to restrict interleavings to I/O-block schedules for programs obeying the ownership discipline, even if the changes of language level occur in a single thread of a concurrent program. Besides the volatile memory accesses, this requires the introduction of additional I/O points: i) at the first step in hypervisor mode ($ASID = 0$) after a switch from guest mode ($ASID \neq 0$) because we need compiler consistency there, and ii) at any step in guest mode because guest computation is in ISA-sp and we no not want to restrict interleavings there. An appropriate general reorder theorem is reported in [40]. Application of the theorem to justify correctness of compilation across the language stack for a version of parallel C-IL without dirty bits and a corresponding simple handling of store buffers is reported in [23].

6.3 Future Work

The same reorder theorem should allow to establish correctness of compilation across the language stack for a structured parallel C-IL with dirty bits down to ISA-u with dirty bits. However, in order to justify that the resulting program is simulated in ISA-sp with store buffers one would need a version of the Cohen-Schirmer theorem for machines with MMUs.

The language stack we have introduced so far appears to establish semantics and correctness of compilation for the complete code of modern hypervisors, provided shadow pages tables (which we introduced in the introduction) are not shared between processors. This restriction is not terribly severe, because modern processors tend more and more to provide hardware support for two levels of translations, which renders shadow page tables unnecessary in the first place. As translations used by different processors are often identical, one can save space for shadow page tables by sharing them among processors. This permits the implemention of larger shadow page tables leading to fewer page faults and hence to increased performance. We observe that this introduces an interesting situation in the combined language stack: the shadow page tables are now a C data structure that is accessed concurrently by C programs in hypervisor mode *and* the MMUs of other processors running in guest mode. Recall that MMUs set accessed and dirty bits; thus both MMU und C program can read *and* write. Now

interleaving of MMU steps and hypervisor steps must be restricted. One makes shadow page tables volatile and reorders MMU accesses of other MMUs immediately after the volatile writes of the hypervisor. To justify this, one has to argue that the MMUs of other MMUs running in guest mode never access data structures other than shadow page tables; with the modelling of the MMU as an explicit piece of concurrent code, this proof becomes part of ordinary program verification.

7 Soundness of VCC and its Use

Formal verification with unsound tools and methods is meaningless. In the context of proving the correctness of a hypervisor using VCC as a proof tool, two soundness arguments are obviously called for: i) a proof that VCC is sound for arguing about pure structured parallel C. ii) a method to 'abuse' VCC to argue about machine components that are not visible in C together with a soundness proof for this method.

7.1 Related Work

In the Verisoft project, the basic tool for proving program code correct was a verification condition generator for C0 whose proof obligations were discharged using the interactive theorem prover Isabell-HOL. The soundness of the verification condition generator for C0 was established in a formal proof [41]. The proof tool was extended to handle 'external variables' and 'external functions' manipulating these variables. Components of configurations not visible in the C0 configuration of kernels (processor registers, configurations of user processes, and device state) were coded in these external variables. The proof technology is described in great detail in [38].

A formal soundness proof for a program analysis tool for structured parallel C is developed in the 'verified software toolchain' project [39].

7.2 Soundness of VCC

An obvious prerequisite for a soundness proof of VCC is a complete specification of VCC's annotation language and its semantics. VCC's annotations have two parts: i) a very rich language extension for ghost code, where ghost instructions manipulate both ghost variables and ghost fields which are added to records of the original implementation language, and ii) a rich assertion language referring to both implementation and ghost data. A complete definition of 'C-IL + ghost' can be found in [22] together with a proof that 'C-IL + ghost' is simulated by C-IL provided ghost code always terminates.

The VCC assertion language is documented informally in a reference manual and a tutorial [42]; the reference manual also has some rudimentary mathematical underpinnings. More of these underpinnings are described in various articles [43–45]. However, there is currently no single complete mathematical definition. Thus establishing the soundness of VCC still requires considerable work (see subsection 7.5).

7.3 Using VCC for Languages Other Than C

As C is a universal programming language, one can use C verifiers to prove the correctness of programs in any other programming language L by i) writing in C a (sound) simulator for programs in L, followed by ii) arguing in VCC about the simulated programs, and iii) proving property transfer from VCC results to results over the original code given in language L. Extending 'C + macro assembly + assembly' programs with a simulator for program portions not written in C then allows to argue in VCC about such programs. A VCC extension for x64 assembly code is described in [46, 47] and was used to verify the 14K lines of macro assembly code of the Hyper-V hypervisor. In the tool, processor registers were coded in a straightforward way in a struct, a so called *hybrid* variable which serves the same role as an external variable in the Verisoft tool chain mentioned above. Coding the effect of assembly or macro assembly instructions amounts to trivial reformulation of the semantics of the instructions as C functions. Calls and returns of macro assembly functions are coded in a naive way. The extension supports gotos within a routine and function calls, but does not support more extreme forms of control flow, e.g. it cannot be used to prove the correctness of thread switch via change to the stack pointer.

Currently, however, there is a slight technical problem: VCC does currently not support hybrid variables directly. We cannot place hybrid variables in ghost memory, because information clearly flows from hybrid variables to implementation variables, and this would violates a crucial hypothesis in the simulation theorem between original and annotated program. If we place it into implementation memory, we have to guarantee that it is not reachable by address arithmetic from other variables. Fortunately, there currently is a possible workaround: physical addresses of modern processors have at most 48 bits and VCC allows up to 64 bit addresses. Thus hybrid variables can be placed in memory at addresses larger than $2^{48} - 1$. Future versions of VCC are planned to support hybrid memory as a third kind of memory (next to implementation and ghost memory) on which the use of mathematical types is allowed; in turn, the formalization of 'C-IL + ghost' should be extended to include this special hybrid memory.

The papers [31, 48] show the soundness of an assembler verification approach in the spirit of Vx86 relative to the mixed 'C-IL + macro assembly' language semantics of our language stack.

7.4 Verifying Device Drivers with VCC

One way to reason about device drivers is to use techniques from concurrent program reasoning. In a concurrent program, one can rarely specify a function on shared state via a pre and post condition on the state of the device, since other concurrent operations may overlap the execution of the function. Instead, one can specify the function as a linearizable operation that appears to take place atomically at some point between invocation of the function and its return. In VCC, the usual idiom for such verification is to introduce a ghost data object representing the abstract state provided by the device in combination with the driver; the state of this object is coupled to the concrete states of the driver and the device via a coupling invariant. Associated with a function call is a ghost object representing the operation being performed; this operation includes a

boolean field indicating whether the operation has completed; an invariant of the operation is that in any step in which this field changes, it goes from false to true and the abstract state of the device changes according to the semantics of the operation. The abstract device has a field that indicates which operation (if any) is "executing" in any particular step, and has an invariant that the abstract state of the device changes only according to the invariant of the operation being performed (which might also be an operation external to the system).

However, another current limitation of VCC is that it allows ordinary C memory operations only on locations that act like memory. This means that it cannot directly encode devices where writing or reading a *memory mapped I/O* (MMIO) address has an immediate side effect on the device state; currently, the semantics of such operations have to be captured via intrinsics.

7.5 Future Work

A proof of the soundness of VCC apparently still requires the following three major steps:

- documenting the assertion language. This language is rich and comprises i) the usual assertions for serial code, ii) an ownership calculus for objects which is used in place of separation logic to establish frame properties, and iii) a nontrivial amount of constructs supporting arguments about concurrency.
- documenting the assertions which are automatically generated by VCC in order to i) guarantee the termination of ghost code, and ii) enforce an ownership discipline on the variables and a flushing strategy for store buffers.
- proving the soundness of VCC by showing i) ghost code of verified programs terminates; thus we have simulation between the annotated program and the implementation code, ii) assertions proven in VCC hold in the parallel C-IL semantics; this is the part of the soundness proof one expects from classical theory (this portion of the soundness proofs for VCC should work along the lines of soundness proofs for rely/guarantee logics – a proof outline is given in the VCC manual [42]), and iii) variables of verified programs obey ownership discipline and code of translated programs obeys a flushing discipline for store buffers; this guarantees correct translation to the ISA-sp level of the multi-core machine.

8 Hypervisor Correctness

Figure 7 gives a very high-level overview of the structure of the overall theory. After establishing the model stack and the soundness of proof tools and their application, what is left to do is the actual work of program verification. Thus we are left in this survey paper with the task to outline the proof of a hypervisor correctness theorem which expresses virtualization of several ISA-sp machines enriched with system calls stated on 'C + macro assembly + ISA-u + ISA-sp'. Fortunately we can build on substantial technology from other kernel verification projects.

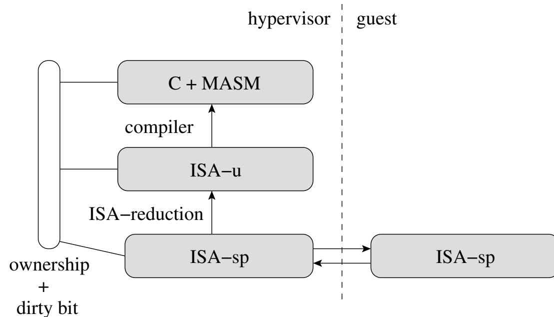

Fig. 6. Using ownership and dirty bit conditions to achieve simulation in a hypervisor model stack by propagating ownership downwards.

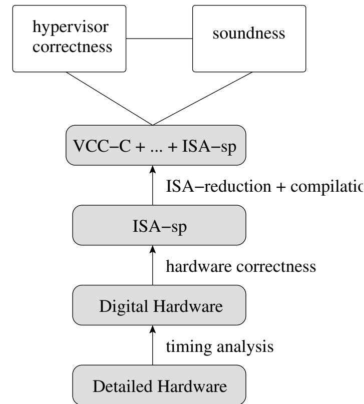

Fig. 7. High-level overview of the model stack and main theorems.

8.1 Related Work

The well known 'seL4' project [49] succeeded in formally verifying the C portion of an industrial microkernel comprising about 9000 lines of code (LOC), although that verification ignored a number of important hardware issues such as the MMU and devices, and used a rather unrealistic approach to interrupt handling, largely because that verification was based entirely on sequential program reasoning. In the Verisoft project [11] both the C portion and the assembly portion of two kernels was formally verified: i) the code of a small real real time kernel called OLOS comprising about 450 LOC [50] and ii) the code of a general purpose kernel called VAMOS of about 2500 LOC [51, 52]. In that project the verification of C portions and assembly portions was decoupled [29] in the following way: A generic concurrent model for kernels and their user processes called CVM (for 'communicating virtual machines') was introduced, where a so called 'abstract kernel' written in C communicates with a certain number of virtual machines $vm(u)$ (see figure 8) programmed in ISA. At any time either the abstract kernel or a user process $vm(u)$ is running. The abstract kernel uses a small number of external functions called 'CVM primitives' which realize communication between the kernel, user processes and devices. The semantics of these user processes is entirely specified in the concurrent CVM model. To obtain the complete kernel implementation, the abstract kernel is linked with a few new functions and data structures, essentially process control blocks, page tables and a page fault handler in case the kernel supports demand paging (e.g. like VAMOS does); CVM primitives are implemented in assembly language. The resulting kernel in called the 'concrete kernel'. Correctness theorems state that the CVM model is simulated in ISA by the compiled concrete kernel together with the user machines running in translated mode. Since the C portions of seL4 are already formally verified, one should be able to obtain a similar overall correctness result by declaring appropriate parts of seL4's C implementation as abstract kernel without too much extra effort.

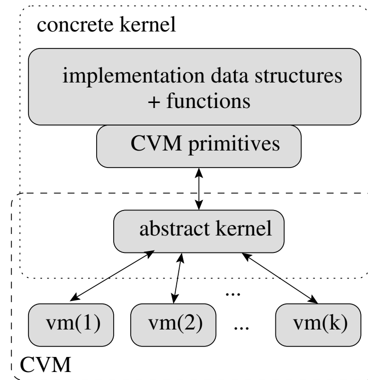

Fig. 8. The CVM kernel model.

8.2 Hypervisor Verification in VCC

That VCC allows to verify the implementations of locks has been demonstrated in [53]. Partial results concerning concurrent C programs and their interrupt handlers are reported in [54]. Program threads and their handlers are treated like different threads and only the C portions of the programs are considered; APICs and the mechanism for delivery of *inter processor interrupts* (IPIs) are not modeled. Thus the treatment of interrupts is still quite incomplete. The full formal verification of a small hypervisor written in 'C + macro assembly + assembly' in VCC using the serial language stack of section 4 (which is also illustrated in figure 6) and the proof technology described in subsection 7.3 is reported in [31, 48]. The formal verification of shadow page table algorithms without sharing of shadow page tables between processors is reported in [23, 55].

8.3 Future Work

The following problems still have to be solved:

- Adding features to VCC that allow memory mapped devices to be triggered by reading or writing to an address that already has a value identical to the data written.
- Proving the correctness of a 'kernel layer' of a hypervisor. In order to provide guests with more virtual processors than the number np of physical processors of the host, one splits the hypervisor in a kernel layer and a virtualization layer. The kernel layer simulates large numbers n of 'C + macro assembly + ISA-sp' threads by np such threads. Implementation of thread switch is very similar to the switching of guests or of user processes. A data structure called thread control block (TCB) takes the role of process control block. Correctness proofs should be analogous to kernel correctness proofs but hinge on the full power of the semantics stack.
- The theory of interrupt handling in concurrent C programs and its application in VCC has to be worked out. The conditions under which an interrupt handler can be treated as an extra thread needs to be worked out. This requires to refine ownership

between program threads and their interrupt handlers. For reorder theorems, the start and return of handler threads has to become an I/O-point. Finally, for liveness proofs, the delivery of IPI's (and the liveness of this mechanism) has to be included in the concurrent language stack and the VCC proofs.
– Proving actual hypervisor correctness by showing that the virtualization layer (which possibly uses shadow page tables depending on the underlying processor) on top of the kernel layer simulates an abstract hypervisor together with a number of guest machines and their user processes. Large portions of this proof should work along the lines of the kernel correctness proofs of the Verisoft project. New proofs will be needed when one argues about the state of machine components that cannot explicitly be saved at a context switch. Store buffers of sleeping guests should be empty, but both caches and TLBs of sleeping processors may contain nontrivial data, some or all of which might be flushed during the run of other guests.

9 Conclusion

Looking at the last section, we see that i) the feasibility of formal correctness proofs for industrial kernels has already been demonstrated and that ii) correctness proofs for hypervisors are not that much more complex, provided an appropriate basis of mixed language semantics and proof technology has been established. It is true that we have spent 6 of the last 7 chapters of this paper for outlining a paper theory of this basis. But this basis seems to be general enough to work for a large variety of hypervisor constructions such that, for individual verification projects, 'only' the proofs outlined in section 8 need to be worked out.

References

1. Verisoft Consortium: The Verisoft-XT Project. URL: http://www.verisoftxt.de/ (2007-2010)
2. Leinenbach, D., Santen, T.: Verifying the microsoft hyper-v hypervisor with vcc. In Cavalcanti, A., Dams, D., eds.: FM 2009: Formal Methods. Volume 5850 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2009) 806–809
3. Hartmanis, J., Stearns, R.E.: On the Computational Complexity of Algorithms. Transactions of the American Mathematical Society **117** (May 1965) 285–306
4. MIPS Technologies 1225 Charleston Road, Mountain View, CA: MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set. 2.5 edn. (July 2005)
5. Freescale semiconductor: Programming Environments Manual for 32-Bit Implementations of the PowerPC™Architecture. (September 2005)
6. Advanced Micro Devices: AMD64 Architecture Programmer's Manual: Volumes 1-3 (2010)
7. Intel Santa Clara, CA, USA: Intel®64 and IA-32 Architectures Software Developer's Manual: Volumes 1-3b. (June 2010)
8. Cohen, E., Schirmer, B.: From total store order to sequential consistency: A practical reduction theorem. In Kaufmann, M., Paulson, L., eds.: Interactive Theorem Proving. Volume 6172 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 403–418
9. Berg, C., Jacobi, C.: Formal verification of the VAMP floating point unit. In: Proc. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME). Volume 2144 of Lecture Notes in Computer Science., Springer (2001) 325–339

10. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Instantiating uninterpreted functional units and memory system: Functional verification of the vamp. In Geist, D., Tronci, E., eds.: CHARME 2003. Volume 2860 of Lecture Notes in Computer Science., Springer (2003) 51–65
11. Verisoft Consortium: The Verisoft Project. URL: http://www.verisoft.de/ (2003-2007)
12. Oracle: Virtualbox x86 virtualization project. URL: http://www.virtualbox.org
13. The Bochs open source community: The Bochs ia-32 emulator project. URL: http://bochs.sourceforge.net
14. The Qemu open source community: Qemu processor emulator project. URL: http://qemu.org
15. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9) (September 1979) 690–691
16. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7) (July 2010) 89–97
17. Sweazey, P., Smith, A.J.: A class of compatible cache consistency protocols and their support by the ieee futurebus. In: Proceedings of the 13th annual international symposium on Computer architecture. ISCA '86, Los Alamitos, CA, USA, IEEE Computer Society Press (1986) 414–423
18. Chen, X., Yang, Y., Gopalakrishnan, G., Chou, C.T.: Efficient methods for formally verifying safety properties of hierarchical cache coherence protocols. Formal Methods in System Design **36** (2010) 37–64 10.1007/s10703-010-0092-y.
19. Paul, W.: A Pipelined Multi Core MIPS Machine - Hardware Implementation and Correctness Proof. URL: http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur2/ws1112/layouts/multicorebook.pdf (2012)
20. Degenbaev, U.: Formal Specification of the x86 Instruction Set Architecture. PhD thesis, Saarland University, Saarbrücken (2011)
21. Baumann, C.: Formal specification of the x87 floating-point instruction set. Master's thesis, Saarland University, Saarbrücken (2008)
22. Schmaltz, S.: Towards Pervasive Formal Verification of Multi-Core Operating Systems and Hypervisors Implemented in C (DRAFT). PhD thesis, Saarland University, Saarbrücken (2012)
23. Kovalev, M.: TLB Virtualization in the Context of Hypervisor Verification (DRAFT). PhD thesis, Saarland University, Saarbrücken (2012)
24. Degenbaev, U., Paul, W.J., Schirmer, N.: Pervasive theory of memory. In Albers, S., Alt, H., Näher, S., eds.: Efficient Algorithms – Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday. Volume 5760 of Lecture Notes in Computer Science. Springer (2009) 74–98
25. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-tso. In: Proceedings of the 24th European conference on Object-oriented programming. ECOOP'10, Berlin, Heidelberg, Springer-Verlag (2010) 478–503
26. Schmaltz, S., Shadrin, A.: Integrated semantics of intermediate-language c and macro-assembler for pervasive formal verification of operating systems and hypervisors from verisoftxt. In Joshi, R., Müller, P., Podelski, A., eds.: 4th International Conference on Verified Software: Theories, Tools, and Experiments, VSTTE'12. Volume 7152 of Lecture Notes in Computer Science., Philadelphia, USA, Springer Berlin / Heidelberg (2012)
27. Leinenbach, D.: Compiler Verification in the Context of Pervasive System Verification. PhD thesis, Saarland University, Saarbrücken (2008)
28. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009) 107–115

29. Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In Hurd, J., Melham, T., eds.: Theorem Proving in High Order Logics (TPHOLs) 2005. Lecture Notes in Computer Science, Oxford, U.K., Springer (2005)
30. Alkassar, E.: OS Verification Extended - On the Formal Verification of Device Drivers and the Correctness of Client/Server Software. PhD thesis, Saarland University, Saarbrücken (2009)
31. Shadrin, A.: Mixed Low- and High Level Programming Languages Semantics.Automated Verification of a Small Hypervisor: Putting It All Together. PhD thesis, Saarland University, Saarbrücken (2012)
32. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with slam. Commun. ACM **54**(7) (July 2011) 68–76
33. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with i/o devices in the context of pervasive system verification. In: 23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2005), 2-5 October 2005, San Jose, CA, USA, Proceedings, IEEE (2005) 309–316
34. Tverdyshev, S.: Formal Verification of Gate-Level Computer Systems. PhD thesis, Saarland University, Computer Science Department (2009)
35. Hillebrand, M., Tverdyshev, S.: Formal verification of gate-level computer systems. In A. Morozov, K. Wagner, A.R., Frid., A., eds.: 4th International Computer Science Symposium in Russia. Volume 5675 of Lecture Notes in Computer Science., Springer (2009) 322–333
36. Müller, S., Paul, W.: Computer Architecture, Complexity and Correctness. Springer (2000)
37. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism
38. Alkassar, E., Hillebrand, M.A., Leinenbach, D.C., Schirmer, N.W., Starostin, A., Tsyban, A.: Balancing the load: Leveraging semantics stack for systems verification. **42, Numbers 2-4** (2009) 389–454
39. Appel, A.W.: Verified software toolchain. In: Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software. ESOP'11/ETAPS'11, Berlin, Heidelberg, Springer-Verlag (2011) 1–17
40. Baumann, C.: Reordering and simulation in concurrent systems. Technical report, Saarland University, Saarbrücken (2012)
41. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
42. Microsoft Research: The VCC webpage. URL: http://vcc.codeplex.com
43. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: Vcc: A practical system for verifying concurrent c. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. TPHOLs '09, Berlin, Heidelberg, Springer-Verlag (2009) 23–42
44. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Proceedings of the 22nd international conference on Computer Aided Verification. CAV'10, Berlin, Heidelberg, Springer-Verlag (2010) 480–494
45. Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A precise yet efficient memory model for C. Electron. Notes Theor. Comput. Sci. **254** (October 2009) 85–103
46. Maus, S.: Verification of Hypervisor Subroutines written in Assembler. PhD thesis, Universität Freiburg (2011)
47. Maus, S., Moskal, M., Schulte, W.: Vx86: x86 assembler simulated in C powered by automated theorem proving. In Meseguer, J., Roşu, G., eds.: Algebraic Methodology and Software Technology (AMAST 2008). Volume 5140 of Lecture Notes in Computer Science., Urbana, IL, USA, Springer (July 2008) 284–298

48. Paul, W., Schmaltz, S., Shadrin, A.: Completing the Automated Verification of a Small Hypervisor – Assembler Code Verification. In: Proceedings of SEFM 2012. Lecture Notes in Computer Science, Springer (To appear)
49. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating system kernel. Communications of the ACM **53**(6) (Jun 2010) 107–115
50. Daum, M., Schirmer, N.W., Schmidt, M.: Implementation correctness of a real-time operating system. In: 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009), 23–27 November 2009, Hanoi, Vietnam, IEEE (2009) 23–32
51. Daum, M., Dörrenbächer, J., Bogan, S.: Model stack for the pervasive verification of a microkernel-based operating system. In Beckert, B., Klein, G., eds.: 5th International Verification Workshop (VERIFY'08). Volume 372 of CEUR Workshop Proceedings., CEUR-WS.org (2008) 56–70
52. Dörrenbächer, J.: Formal Specification and Verification of a Microkernel. PhD thesis, Saarland University, Saarbrücken (2010)
53. Hillebrand, M.A., Leinenbach, D.C.: Formal verification of a reader-writer lock implementation in c. Electron. Notes Theor. Comput. Sci. **254** (October 2009) 123–141
54. Alkassar, E., Cohen, E., Hillebrand, M., Pentchev, H.: Modular specification and verification of interprocess communication. In: Formal Methods in Computer Aided Design (FMCAD) 2010, IEEE (2010)
55. Alkassar, E., Cohen, E., Kovalev, M., Paul, W.: Verification of tlb virtualization implemented in c. In: 4th International Conference on Verified Software: Theories, Tools, and Experiments, VSTTE'12. Lecture Notes in Computer Science, Philadelphia, USA, Springer-Verlag (2012)