Translating the partitioned code The most basic idea for translating the now partitioned basic blocks is to have a fixed association that maps every instruction in the guest ISA to a sequence of instructions native to the host.

The quality of the code that can be generated here strongly depends on the properties of the host and guest architectures in question. Difficulties can arise due to differences in the instruction operand formats and the type of instruction set architecture the DBT is dealing with.

In our case, as outlined in section sec:isa-cmp, challenges stem from the fact that we are translating code from a load-store architecture using a three-operand instruction format into a register-memory architecture in which (generally) one of the source operands is also the implicit destination operand. This, for example, means that a single arithmetic add rd, rs1, rs2 in RISC-V assembly language generally can not be translated via a single instruction, but rather requires two instructions: moving rs1 to rd, then adding the value of rs2 to rd.

Opportunities for optimisation lie wherever there is a way to shorten the translation's amount of CPU clock cycles, possibly by employing semantically equivalent native instructions that run in a shorter timespan. The RISC-V pseudoinstructions (as mentioned in section sec:isa-cmp) are also of some help here [S. 139]riscvspec, along with discoverable patterns in the input assembly. It is clear, for example, that an instruction like xori x10, x10, -1 can be directly translated as a not x10, without needing to resort to mov and xor. The same principle applies to combinations of multiple instructions. An lui rd, imm1 followed by addi rd, rd, imm2 may for example be translated as directly loading the result of the computation imm1 + imm2 into rd.

Code cache and block handling Naturally, the DBT aims to store the translated code in a semi-permanent way, for it is the goal to not have to translate a required section more than once.

For that, we allocate a region of memory reserved for the basic block translations, also called a code cache. Additionally, an index to this memory section is required, since there needs to be a way to quickly reference the blocks residing in the cache and associate them with both the host and guest instruction pointers that identify them during execution.

It is possible that this code cache might fill up during the execution of a large guest program. If it does, there are two different strategies to handle this issue: One can either invalidate and purge some or all of the blocks currently residing in the cache, or dynamically resize the cache according to the needs of the guest program [S. 3]bintrans.

Purging the entire cache would require the translator to restart translation on older blocks that might be needed again, introducing a performance overhead that needs to be weighed against the higher memory usage of enlarging the cache.

On the other hand, selective deletion of some of the blocks in the cache is very difficult due to optimisations taken in the context of chaining. As any chained jumps located in another cached block are dependent on the target block residing in the cache, the target's removal would invalidate these jumps. It would thus only be possible to either remove all blocks with jump references to the candidate up for removal, or to leave all blocks with jump references in the cache altogether.

Register handling and context switching sec:context-switch-reg-handle

Handling of guest registers As outlined in section sec:isa-cmp, the RISC-V and x86-64 architectures have differing amounts of general purpose registers. In some way, the state of the 32 general purpose registers x1x0 is hardwired to a constant zero. All reads will return 0, all writes will be ignored. Hence, this register needs special handling in the DBT, as there is no equivalent construct on x86-64.o x31 and the pc needs to be stored and available to the translations of the identified basic blocks.

As x86-64 only provides for 16 general-purpose registers (rax–rdx, rsi, rdi, rsp, rbp and r8–r15), it is impossible to directly and statically map all guest registers to native host registers. Adding to the above, due to the fact that some x86-64 registers have special or implicit purposes in some instructions like (i)mul or (i)/div, care must be taken in choosing the registers that can be used for such a mapping. Keeping a guest register file exclusively in memory, and loading them into native registers when needed within the translations of single instructions is technically possible, especially in light of x86-64's ability to extensively use memory operands in the instructions. However, this necessitates a large number of memory accesses for

construct on x86-64. t

both memory operands in the instructions as well as local register allocation within the translated blocks. Due to the very large performance gain connected to using register operands instead of memory operands, this is also not feasible at scale [S. 8f.]bintrans.

Accordingly, the solution for this problem would be an approach that employs parts of both of these extremes [S. 9]bintrans. We utilise the tools we designed to discover the most-used registers in the guest programs, and statically map these to general purpose x86-64 registers. The remaining operands are then dynamically allocated into reserved host registers inside a single block's translation. The loaded values are then lazily kept in the temporary registers for as long as possible in order to avoid unnecessary memory accesses. In case the translator requires a value not currently present in a replacement register, the oldest value is written back to the register file in memory and the now free space is utilised for the requested value. The final write-backs then need to be performed on the block boundaries.

The most-used registers are relatively invariant in between RISC–V executables and their basic blocks, however it might be the case that a single block in such an executable requires a few unique registers fairly often. By dynamically allocating these into temporaries and statically mapping the most-used registers in general, we save much overhead otherwise spent on memory access to the register file, but do not unnecessarily occupy native register space with seldomly accessed guest registers.

Context switching during execution When the code translated by the DBT is executed, it will behave as if it were an independent x86–64 executable. With the static register mapping in place, these values will thus need to be loaded before any of the translated blocks are called, and stored back before the execution is returned to the DBT.

This is called a context switch, as we are switching from the host's program state made up of the current register values to that of the guest. Evidently, preserving both the host and guest's state during execution is critical for the correct program behaviour.

System call handling sec:syscall-handling System calls are also a very important part of enabling the guest program's execution. Thus, every ISA must offer some way to switch the execution context in the kernel mode for the system call to be handled.

For RISC-V, the instruction ECALL (for environment call, formerly SCALL) handles these requests, with the system call number residing in register a7 and the arguments being passed in a0–a6.

However, the DBT generally cannot just reorganise the guest argument values and system call identifier according the host's calling convention and relay the system call directly. The RISC-V guest program expects a different operating system kernel than is present natively on the host; with that, the system call interface also differs [S. 2f.]bintrans.

In order to handle the ECALL instruction correctly, the translator must thus build the translated instruction to call a specific handler routine not too dissimilar from one that may be found in a kernel. There, system calls that exist natively on the host architecture as well (like write or clock_gettime) can usually be passed along to the host's kernel directly.

Care must be taken for system calls that would enable the guest to change the state or context of the host – an mmap into the translator's memory region, for example, or a call to exit – these calls must be emulated accordingly to prevent these faults. In cases where the data structure layout used by the kernels differs, the DBT must also perform necessary actions to adapt the formats to each other. Some system calls may not exist at all on the native architecture of the host, it is up to the DBT to emulate the required functionality [S. 2f.]bintrans.

Floating point extension subsec:fp$_{e}xtensionFloatingpointsupportisavitalpartofmodernprocessors, enablingfastcon$
$64SSEextensionisevidentlyalotfaster.$

The main difficulties (and their resolutions) that arise by using the x86-64 SSE extension to translate the RISC-V F- and D-extensions are listed below: itemize

Register handling is similar to the integer register management laid out in section sec:context-switch-reg-handle. As mentioned before, the RISC-V architecture consists of 32 floating point registers (f0 – f31) which can hold a single precision (F-extension) or double precision (D-extension) floating point value, whereas the SSE-extension only provides 16 registers XMM0 – XMM15. We utilise the tools we designed to discover the most-used registers in the guest programs, and statically map these to x86-64 SSE registers XMM2-XMM15. One could use the same dynamic mapping approach for the remaining registers as is being used for general purpose registers, but for simplicity register XMM0, XMM1 are

just reserved as replacements and missing registers are moved into them from memory temporally.
Missing equivalent SSE instruction can lead to a huge instruction overhead, as emulation often use bit manipulation operations instead. Therefore constants or masks need to be moved in from either memory or the general purpose registers because the SSE extension does not support immediates. The instructions that need to be emulated are unsigned conversion instructions e.g. FCVT.WU.S, sign-injection instructions e.g. FSGNJ.S, compare instructions e.g. FEQ.S, fused multiply-added instructions e.g. FMADD.S and the FCLASS.S instruction that classifies a floating point value. These instructions are not supported by the SSE extension natively. As an implementation reference for these instructions, the assembly generated by gcc using the Godbolt Compiler Explorer godbolt was used.
Rounding modes are handled differently in the RISC-V architecture, as the rounding mode can be set individually for every instruction. The rounding mode of the SSE extension however is controlled by the state of the MXCSR control and status register. One could use EVEX-encoding [S. 374]intel2017man to set the rounding mode on a per instruction level for x86-64 as well, but faenc faenc does not support EVEC-encoding so far. Thus in case a instruction with explicit rounding mode is encountered, the rounding mode is temporally changed in the MXCSR register.
Exception handling in RISC-V is realized by reading the fcsr floating-point control and status register, traps are not supported. The CSR instructions used to read this register are thus emulated to instead read and translate the MXCSR exception flags. x86-64 exceptions are meanwhile disabled by masking them in the MXCSR register.