Arancini Hybrid Binary Translator: Design of the Dynamic Binary Translators for ARM64 and RISC-V

Theofilos Augoustis
Technical University of Munich

Abstract

Arancini is a Hybrid Binary Translator (HBT) for x86 to the ARM and RISC-V architectures, aiming at achieving high performance ISA emulation through extensive use of compiler-based techniques. It utilises a Static Binary Translator (SBT) based on LLVM that converts machine code to a target ISA, applying optimizations and inserting a runtime emulator component. Limitations of the SBT for the x86 input architecture are handled via the use of an architecture-specific Dynamic Binary Translator (DBT), which is invoked at runtime for some basic blocks.

This paper describes the design and implementation of the ARM64 DBT, its interface to the rest of Arancini and its handling of various problems in ISA emulation.

1 Introduction

The increasing popularity of ARM in desktop computing and server computing, as well as the rapid developments in the RISC-V platform, have led to a renewed interest by OEMs in RISC-based computers. In particular, Apple has developed and utilised the M1 ARM-based CPU in their latest laptops.

A traditional problem in popularising a new platform has been porting existing software to the new ISA. Large amounts of existing software developed for Windows, Linux and even MacOS is available only for x86 CPUs. The availability of software is a determining factor in the success of a new computing platform [6]. An approach for enabling the use of software for other platforms on a new system is virtualization, which has been used extensively by Apple with their Rosetta software [9].

A widely used virtualization technique for such purposes is process-level ISA emulation, which translates the instructions of a binary in a source ISA to one or more instructions in a target ISA, while exposing virtualized OS services using the host OS. This can be achieved by translating instructions at runtime using a DBT, translating the entire program before execution using an SBT or using a hybrid of both approaches with an HBT.

A Static Binary Translator (SBT) translates the entire machine code to the target ISA before program execution, yielding an equivalent program in the target ISA, which may be further optimized using a variety of compiler-based methods. However, SBTs have major limitations due to the code-discovery problem and the code-location problem [13]. In particular, code may be interspersed with (padding) data that is indistinguishable from instructions, instructions have variable-length encodings in CISC ISAs that makes decoding them dependent on the previous instruction and indirect-branch target addresses cannot always be determined at runtime [2]. In fact, given a register-indirect branch, determining the value of the register for calculating the target address is equivalent to resolving the halting problem [7]. At the time of writing, there exist no fully-functional SBT implementations for the x86 ISA to any target ISA.

The traditional limitations of SBTs are avoided by DBTs, which include a runtime component that translates machine code during execution. DBTs operate on the principle of reading instructions in the source ISA, interpreting them or translating them in basic blocks and executing the translated code. Since the actual program is executed, control flow and branch target addresses are determined at runtime, so that the relevant machine code can be discovered fully. Furthermore, the existence of a runtime component allows for tracking various state, such as the mapping between the source program counter and the target program counter, avoiding the code-location problem [13]. However, this introduces runtime overhead due to the DBT software, which can be relatively large. Despite the overhead, DBTs are used extensively in the design of process-level VMs, such as QEMU [3].

A major selling point of any new processor architecture used in desktop and server computing is performance [6]. However, existing DBT-based emulators have large performance overhead, which is assumed for each execution of the program [4]. As a result, HBTs have been gaining popularity for high performance emulation, as evidenced by Rosetta 2 developed by Apple [9] and academic research on the subject [12]. An HBT utilises an SBT for parts of the machine code that can be translated statically, using control-flow analysis to determine the location of machine code and compilation to produce equivalent code in the target ISA [12]. Due to limitations of SBTs, there will be machine code that is not discovered at compile-time and branches that cannot be determined, in which case the DBT will execute those parts.

Arancini aims to implement an HBT utilising LLVM, its own IR (called IRancini) and a number of custom-designed DBTs. It can handle a wide variety of x86 programs, which it lifts to its own IR and translates using a backend. The LLVM backend requires raising IRancini to LLVM IR, which then handles the static translation. The DBT backend converts IRancini to the target ISA specific code, which is executed through the runtime interface. In particular, Arancini can emulate programs using either hybrid translation or just a DBT. Note that compared to Rosetta 2 [9], Arancini is free and open-source.

Overview

Arancini consists of a set of input translators, translator backends and a runtime component. The components manipulate IRancini to perform the translation, which is managed by the runtime. A translator component (called *txlat*) is provided as the interface to Arancini, it is invoked to perform the translation on a user-provided x86 binary.

Input Translators

This is the starting point for the translation of the input architecture. For each instruction in the input machine code, it uses a factory pattern to get a translator specific for that category of instruction, which is then used to translate the instruction to IRancini. Each instruction is translated into a packet, which is then added to the output chunk.

IRancini

The Arancini IR represents instructions using *nodes* that define *ports* for their inputs and outputs. Multiple types of nodes are defined, corresponding to some conceptual operation performed in an instruction. Each instruction is considered as having multiple nodes, connected in a graph with other nodes through their ports, as seen in

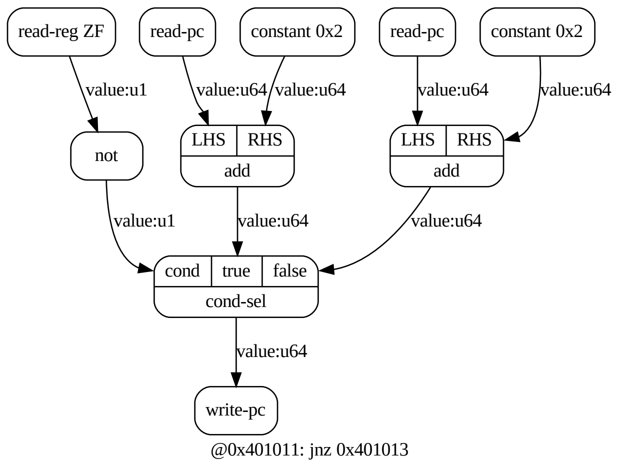

Figure 1: Sample IRancini for Jump-if-Not-Zero

Figure 1. As such, nodes can be combined together to fully represent the behaviour of x86 instructions.

There are two main categories of nodes: *action nodes* and *value nodes*. An *action node* represents an operation with side-effects, such as writes to a register, and it is used to form a tree of *value nodes* that determine its inputs. A *value node* represents an operation without side-effects, such as reading a register.

In contrast to traditional compiler IRs, IRancini does not aim to encode the CFG of the program, but rather its layout in the binary. It is constructed through the input translators from the x86 machine code of the program and it is consumed by all backends.

Backends

There are two major backend types, a static backend (for the SBT) and a set of dynamic backends (for the DBTs).

The static backend consists of an engine that converts IRancini to the LLVM IR, while preserving its semantics. It also inserts the interface code into the runtime DBT component for parts of the IR that could not be lifted to LLVM IR.

The dynamic backends consist of the x86, ARM and RISC-V DBTs. They are provided as shared libraries, linked to the program as part of its runtime component.

The x86 DBT converts the IRancini to x86 code, it is used for showcasing the expressiveness of the IR and as a baseline for performance measurements.

The ARM and RISC-V DBTs are used for translating IRancini to aarch64 ISA ARM and RV64G code in RISC-V, respectively. The ARM DBT is described in detail.

Runtime

The Arancini runtime component handles the source program code, the register state of the source ISA in a context block, the emulated process memory, multithreading and the interface code with the DBT translation.

The interface to the code generated by the DBT is handled using an architecture-specific *trampoline*. The runtime code relies on the trampoline to save existing state, set registers as expected by the translation and restore register state after translation.

Translator

The translator component is the user interface to Arancini. It consists of *txlat*, an executable that uses the Arancini runtime library to translate an executable.

It can produce an equivalent executable in the target ISA using static translation, but includes additional code sections for the runtime component. The executable produced can then be directly executed by a user without additional configuration. Furthermore, it can similarly produce an executable that includes the DBT code only, which can be executed in the same way.

The translator supports configuration through program options, which enable various settings meant for debugging. It supports dumping information about the translation process, including debug information in the produced binary and producing a DOT graph-view of the IRancini for the original program.

Note that the translator internally invokes a compiler, so Arancini depends on a compiler toolchain being available on the host system.

DBT Design

The Arancini DBTs are implemented to perform translation at the level of a dynamic basic block. A dynamic basic block is similar to a basic block, it contains only a single exit point, but can contain multiple entry points [13].

The runtime component of Arancini iterates through source instructions at runtime, translating them to IRancini using an input translator. The resulting IRancini is iterated again, the DBT being invoked for each action node, producing machine code for the target ISA. The resulting machine code is saved to a code cache and executed through an architecture-specific trampoline.

The runtime maintains a cache of translations, indexed using the source program counter of the first instruction in the dynamic basic block. This cache is searched before performing any type of translation, as a performance optimization. The code cache size and replacement policy are important design considerations, since they have a direct effect on performance [13]. An important difference in Arancini is that the DBT may operate on a relatively small number of dynamic basic blocks, since the bulk of the source code is expected to be translated by the SBT, making the code cache size less of a concern.

Either way, when the machine code has been determined for a given block of instructions, it must be branched to and executed. This is performed through the trampoline, which saves existing state, sets specific registers to point to the x86 context block (containing among others the x86 program counter), executes the translated block and then restores the state after completion. The main information needed by the translated instructions is the context block for reading register values and the guest memory base to address memory.

DBT Implementation

The DBT is represented by a *translation context*, which provides an interface consuming IRancini. The beginning and end of basic blocks and instructions are specified by the runtime component, while the `lower()` method is invoked for lowering an IRancini node to the DBT. Correct usage of this interface iterates through IRancini action nodes and calls `lower()` on each of them.

Internally, the DBTs are organized in terms of a set of `materialise*()` functions. There is a function for each node, called from a master function that determines the node type. The pseudocode for master function is shown abbreviated in Figure 1.

```
void materialise(node) {
    // Skip node if already materialised
    if (materialised_nodes.contains(node))
        return;

    switch (node.kind()) {
    case node_kinds::constant:
        materialise_constant();
        break;
    case node_kinds::binary_arith:
        materialise_binary_arith();
        break;
    ...
    ...
    default:
        throw runtime_error("Node does not
            exist");
    }
}
```

Listing 1: Listing for the materialise() function

Each `materialise*()` function follows a similar pattern internally, it invokes recursively `materialise()`

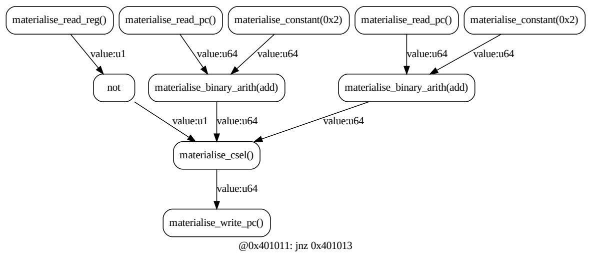

Figure 2: Sample calls to *materialise()* for Jump-if-Not-Zero

for each of the input ports of the node and then proceeds with implementing the actual functionality of that node. This behaviour is illustrated in Figure 2, which closely matches Figure 1. An important mechanism implemented through the calls to *materialise()* and its associated data structures is tracking dependencies on values produced. These dependencies are used to construct an internal representation of an instruction, which is then used to produce machine code for it. This is done by interacting with an *assembler* that produces machine code for that node.

Each DBT interacts with a different assembler library for converting the instruction representations manipulated by the DBT to machine code. The ARM DBT relies on the Keystone library for the assembly, passing to it strings containing assembly instructions. Keystone provides the `ks_asm()` interface that supports passing an entire *stream* of instructions, such that assembler calls can be minimized. As such, the DBT will progressively append instructions to a stream as it processes the instruction block and assemble them in one go.

A *materialise()* call for an input port generally leads to values being generated, which must be communicated to the caller of the function. This is done using *virtual registers*, which can be considered as variables for the purposes of the DBT. A *port-to-virtual-register* data structure is maintained, alongside a monotonically increasing counter that represents the last virtual register assigned. The number of virtual registers is conceptually unbounded, the only limits is that it must be representable within a 64-bit value. This is considered sufficient, since dynamic blocks do not generally contain so many instructions as to produce an overflow. Each `materialise*()` function calls a `alloc_vreg_for_port()` helper that increments the virtual register counter and assigns it to the port map. The recursive calls needed for each input port will likewise insert into the port mapping, while top-level functions will reference the mapping to retrieve the virtual register index. It is also possible to use a virtual register without assigning to any port, which is useful for tracking dependencies *within* one `materialise*()` call. This is needed when there are multiple instructions generated for a single port that have dependencies. Both approaches are shown in Figure 2.

```
void materialise_read_mem(read_mem_node n) {
        int dest_vreg = alloc_vreg_for_port(n.
            val());
        int w = n.val().type().element_width();


    // Add base to addr_vreg
    int addr = alloc_vreg();
    auto addr_op = vreg_operand_for_port(n.
        address());
    builder_.add(virtreg_operand(addr, 64),
                memory_base_reg, addr_op);

    auto mem = arm64_memory_operand(
        arm64_vreg_op(addr, w));
        builder_.ldr(virtreg_operand(dest_vreg,
             w),
                arm64_operand(mem));
}
```

Listing 2: materialise() for read memory nodes

Although the model presented in Figure 2 works for most nodes, it has limitations for forward branch instructions. This is because the branch target label is set only when the target node has been generated, such that they cannot be materialised immediately when encountered. The ARM DBT avoids this problem by buffering nodes passed to `lower()` and performing the actual translation by iterating over all buffered nodes when the end of the basic block is reached. IRancini makes a distinction between branches that occur in the representation of an instruction and those that happen between multiple source instructions. The former are represented as branch nodes, while the later are implemented as reads followed by writes of the program counter.

Note that in the case of branches between source instructions, the ARM DBT requires that the program counter of the current instruction be available. This is possible because it is passed by the runtime component to the DBT for every instruction.

The use of virtual registers allows for postponing register allocation during translation of IRancini to machine code, such that the design can be relatively simple. This separation also enables the implementation of an optimization step, but does require implementing explicit register allocation logic. In effect, this is similar to the approaches taken by compilers for code generation.

Instruction Representation

The ARM DBT uses a set of data structures for representing instructions, operands, labels and dependencies. These data structures are ultimately used to create a string representation of each instruction or label, but allow the DBT to interact with objects instead of manipulating the string characters.

Each instruction is represented by an `arm64_instruction`, which contains an array of `arm64_operand` objects and an opcode as a string. The main method provided by `arm64_instruction` is `dump()`, which generates a string representation of that instruction. The same method is provided by `arm64_operand`, such that `arm64_instruction` can simply call `dump()` for each operand of the instruction. Although the interface is simple, this elegantly transforms the class-based instruction representation into a string.

The operands are handled by the DBT based on their type, which can be any among:

- physical register
- virtual register
- immediate
- memory operand
- shift
- label
- condition

Each are represented by a specific class, which is used to initialise `arm64_operand` types. Each class contains information required to generate a string representation of the type. For example, `arm64_physreg_op` will contain an index into a table of aarch64 register names, while a `arm64_memory_operand` will contain either a `arm64_vreg_op` or a `arm64_physreg_op` (physical register operand), an offset and flags for pre- or post-indexing. The `dump()` code differentiates between representations and matches them with ARM assembly patterns.

The actual DBT `materialise*()` functions will create different operand types and assign them to the instruction that they are translating. Although converting to string representations requires additional complexity, this is only performed right before invoking the assembler and other operand manipulations are done on simpler C++ classes.

This operand model fits nicely with the need for virtual registers during translation, as they represent just another operand type. Likewise, labels that are sometimes needed by instructions (e.g. to implement branches) are also operands and so are shifts. However, there are no restrictions imposed on the operands and the opcode set for the instruction, incorrect instruction specifications are caught by the assembler and returned as an error.

Although this model can represent any aarch64 instruction, there is also a need to be able to specify labels or even comments. This is simple to implement with existing types, an `arm64_instruction` without operands is created with the opcode set to the label or comment string. This approach treats labels, comments and actual instructions transparently, relying on the assembler to detect errors and the DBT to correctly create `arm64_instruction` objects.

Translations

The ARM DBT is able to translate basic programs compiled for the x86-64 ISA, including hello world linked to a minimalist LIBC implementation provided by musl [1]. It includes functionality for most basic x86-64 instructions and can emulate some system calls using equivalent system calls of the host OS. Since there exist x86-64 ISA extensions that are not handled by the ARM DBT, Arancini is only able to provide *extrinsic compatibility* in these cases [13].

There are multiple problems with emulating x86-64 instructions on ARM, as a result of differences in CISC vs. RISC design philosophies and due to the inherent need to virtualise the system environment [2]. Those issues include handling of x86 register state, memory state, the stack and user-visible divergence from the expected behaviour. In particular, the state of the x86 registers must be maintained to ensure correct execution, including the flag registers that are used to determine the behaviour of conditional branches [13].

The approach taken by the ARM DBT is to use a *context block* that contains the state of the x86 registers, which is kept up to date by the SBT. The DBT relies in its translation on the context block, each instructions reads its source operands from the context block to the ARM registers and writes its results to the x86 registers in the context block upon completion. This requires multiple memory accesses for each instruction and comes with an expected significant performance overhead. However, this approach is simple and ensures that the x86 context block is kept up to date at any moment in time, which simplifies implementing precise exception handling [6]. This behaviour can be seen in Figure 3.

Another design consideration is the handling of the flags presents in the *rflags* register of x86-64. These flags are set by binary arithmetic operations, writes to them are tracked via IRancini and the DBT can detect them when handling `write_reg` nodes. It is important to note that ARM64 also includes flags as part of the ISA, which are likewise set by various binary arithmetic and comparison operations. The approach taken by the ARM DBT is to conditionally set x86 flags depending on the values of the ARM flags and update them in the context block. This introduces additional overhead, compared to an approach such as *state mapping* [13], but it is also simpler to implement and easier to debug. Figure 3 contains the `cset` instructions that manipulate the flags.

There exist some differences in the handling of immediate values between aarch64 and x86-64, since aarch64 instructions accept only immediates with smaller representations. This is particularly needed for branch instructions on x86, which encode their branch target address as an immediate. The approach taken by the ARM DBT is to handle such immediates by determining the number of bits required for their representation and writing them to registers with multiple shifted `mov*` instructions. A `mov*` instruction accepts an immediate values representable within 16 bits, meaning that up to 4 `mov*` instructions are needed to load a 64-bit immediate. This is likewise shown in Figure 3.

The ARM DBT includes functionality for dumping translations for each dynamic basic block before its assembly and execution, meant to be used for debugging DBT behaviour. This allows for analysing translations for correctness and reveals patterns in generated code.

```
// Load from context block
ldr x0, [x29, #0x28]

// Load 64-bit immediate into x1
movz x1, #0xfff0, LSL #0x0
movk x1, #0xffff, LSL #0x10
movk x1, #0xffff, LSL #0x20
movk x1, #0xffff, LSL #0x30

// Perform the logical AND operation
and x2, x0, x1

// Conditionally set Zero flag
cset x1, eq

// Conditionally set Sign flag
cset x0, lt

// Store result of AND
str x2, [x29, #0x28]

// Update Zero flag
// Note: register-indirect access
mov x2, #0x888
add x2, x29, x2
str x1, [x2, #0x0]

// Update Carry flag
// Set to 0
mov x1, #0x0
mov x2, #0x889
add x2, x29, x2
str x1, [x2, #0x0]

// Update Overflow flag
// Set to 0
mov x1, #0x0
mov x2, #0x88a
add x2, x29, x2
str x1, [x2, #0x0]

// Update Sign flag
mov x1, #0x88b
add x1, x29, x1
str x0, [x1, #0x0]
```

Listing 3: Produced aarch64 assembly for: AND %rsp, $0xFFFF\ FFFF\ FFFF\ FFF0$

Register Allocation

The register allocator used by the ARM DBT is a reverse *linear scan* allocator [10]. The allocator executes at the level of a dynamic basic block, after a translation for it has been generated. The basic algorithm followed traverses instructions in the translated dynamic basic block using reverse iterators, allocating registers as needed to each instruction. The allocation itself relies on dependency information for each instruction operator, which is provided during translation by the ARM DBT.

The functionality of the register allocator is encapsulated within the `arm64_instruction_builder` class, which tracks both the translation for a given dynamic basic block and provides the method `allocate()` for performing the register allocation. The actual dependency information is provided by the `arm64_operand` class with methods `use()`, `def()` and `usedef()`, which mark an operand as being used, defined or both by an instruction.

A critical design decision for the register allocator is to separate its functionality from the actual translation, achieved through the use of virtual registers. This is in contrast to the approach taken in the RISC-V DBT and follows the design philosophy of compilers, which generally perform register allocation as a discrete step of compilation. The benefits are similar, the translations can be optimized independently of the register allocation, the allocation itself can be performed only on the optimized code and generally the register allocator can be ported easily.

The register allocator maintains a 32-bit bitmask that represents register allocations. The bitmap is initialised to allow any ARM register to be allocated, except the SP/Zero (Stack Pointer), the FP (Frame Pointer), the X30 register and register X18. The SP/Zero register cannot be written to directly by instructions, while the FP register is used as the base for the context block. Registers X30 and X18 contain the return address to the trampoline code and the memory base for main memory accesses, respectively. The remaining registers are assigned by the register allocator and deallocated as needed. Note that although the remaining 28 registers are almost double in number compared to those of x86, register spilling is still theoretically necessary, since each x86 instruction translates to multiple ARM instructions.

The actual register allocation is divided into two stages for each instruction, with virtual register definitions being mapped to physical registers first and then used virtual registers being mapped after.

The process of handling virtual register definitions involves a *virtual-to-physical-register* map, which is updated when a virtual register used by an instruction is mapped to a physical register. The definitions check the mapping to verify if there are any users of the value produced by the definition, in which case it assigns the virtual register to the physical register given by the mapping and also makes the register available in the register bitmask. This bitmask setting is equivalent to releasing the register, which is possible as the instruction writes to it and previous values do not impact execution after this point. If there are no users of the defined value, the instruction is discarded. The mapping is guaranteed to contain an entry for a virtual register if it is used by a later instruction, since instructions are traversed in reverse order, starting from the last in a dynamic basic block. This behaviour is desirable as an optimization for eliminating unnecessary instructions.

The used virtual registers are allocated by checking the virtual-to-physical-register map. If there already exists a mapping for a virtual register (e.g. because a later instruction uses the same register to denote a dependency on the value), the virtual register is allocated to that physical register. Otherwise, the register bitmap is checked and the first available register is used to perform the allocation. It is worth noting that at this point, depending on the availability of physical registers in the bitmask, the register spilling procedure can be invoked to release some registers. The same process must be followed for main memory operands, which may use virtual registers to perform register-indirect accesses. However, note that this cannot occur in the definition phase, as main memory operands cannot be definitions due to the RISC-like design of the ARM ISA.

The state of the register allocator is cleared on each new dynamic basic block, since basic blocks are handled independently of each other. In the current implementation that does not perform *state mapping* of ARM to x86 registers, the translation for each instruction will use some number of ARM registers and the register allocator will implicitly release all of them before processing them next instruction. This is because the actual dependencies between x86 instructions are handled via the context block, instead of the model employed above.

System Calls

Arancini is a process-level VM, it must provide to the emulated program the same facilities available to any other process on the system. In order to achieve this, it must support system calls, allowing the process to interact with the host kernel. However, it must provide a translation layer, since procedures for invoking system calls differ between architectures on the same kernel and might require emulation to handle various possible corner cases [5].

This is achieved using the *internal call* mechanism in Arancini, which is used to represent and handle system calls. This mechanism consists of a section of code, represented by `internal_call()`, which matches source ISA system call numbers to an implementation routine. This implementation routine generally consists of a call to the actual system call, but may also include additional code for state handling, address translation (for `ioctl()`), etc.

The `internal_call()` section must generate code that invokes a system call for most system calls requested by the emulated process. The machine code sequence differs among architectures, while the actual emulation in `internal_call()` is portable. Arancini uses a `native_syscall()` procedure for abstracting the differences in system call handling among platforms. The implementation of `native_syscall()` for a system call with 2 parameters is shown in Figure 4, similar implementations exist for more parameters. The same approach is taken in the RISC-V DBT.

```
template <> inline uint64_t
native_syscall(uint64_t syscall_no, uint64_t
    arg1, uint64_t arg2)
{
        register uint64_t x0 __asm__("x0") =
            arg1;
        register uint64_t x1 __asm__("x1") =
            arg2;
        register uint64_t x8 __asm__("x8") =
            syscall_no;
        __asm__ __volatile__("svc #0\n\t"
                        : "=r"(x0)
                        : "r"(x8), "0"(x0), "r"
                            (x1)
                        : "memory");
        return x0;
}
```

Listing 4: `native_syscall()` implementation with 2 parameters

The `internal_call()` section is inserted into the translated binary using runtime component, which uses the interface provided by `llvm::BasicBlock` to generate it in the binary. However, it must also be exposed to the static and dynamic backends, such that they can notify the runtime that a system call must be executed. The DBT interface achieves this using return values, which distinguish between successful completion of a translated dynamic basic block, a basic block that ends in a system call or an erroneous condition that must cause emulation to stop.

The ARM DBT always generates an instruction that sets the first register used for return values to an appropriate immediate. The following returns are recognised:

1. 0 - successful completion
2. 1 - system call (system call number stored in rax)
3. 2 - erroneous condition (abort execution)

The runtime component uses two conditional branches for checking the return code, branching to the main loop block to continue with the next basic block, to the `internal_call()` basic block or to a special exit block for aborting execution.

It is worth noting that the current implementation of the ARM DBT only handles a limited number of system calls, sufficient to run a hello world program and the Phoenix benchmark [11].

Development Considerations

Arancini is inherently multi-platform software and must depend for its core functionality on relatively large libraries, like LLVM. This introduces various problems that must be handled at the level of the build system and requires a multi-platform development environment.

The build system employed relies on CMake for fetching dependencies, including XED, FADEC and Keystone. It also relies on CMake for finding dependencies before fetching, such as LLVM. The ability to interface with other build systems and its relative popularity makes CMake the ideal candidate. As an added benefit, CMake can handle cross-compilation via the definition of a toolchain file, which is provided for ARM64.

However, even with build system support for cross-compilation, there is still a need to configure or provide dependencies. This has led to large compilation times in our implementation, given that LLVM and XED are both large libraries. As such, the following setups were used for development:

- Native ARM, x86 and RISC-V setups for native development
- Docker containers with multiplatform support via *binfmt*
- Docker-based environments for cross-compilation
- QEMU-based VMs for ARM and RISC-V
- File mirroring for remote development

Future Developments

The topic of DBTs has been extensively researched and there exist multiple techniques that can provide higher performance. Existing emulators such as QEMU that rely on DBTs [3] and even language VMs (e.g. JVM) have adopted such research, sometimes achieving superior performance to native runtimes. On the other hand, the HBT setup may rely on the DBT for only small proportions of the overall program code. Furthermore, even if those proportions corresponding to hotspots, the use of caching avoids high overhead as a result of the DBT. Considering Amdahl's Law [6], additional optimization in the DBT may be unnecessary.

An optimization that may yield significant improvements to performance is *basic block multi-versioning* [13] using the static backend. This method would gather usage data for basic blocks translated by the DBT, allowing it to determine hotspots. Once a hotspot has been determined, it would use a *background thread* for invoking the static translator for the basic block with optimizations turned on. When the static translation has completed, the translated basic block by the DBT could be replaced by the optimized translation produced by LLVM, hence achieving superior performance.

Another optimization would be *state mapping* of x86 to ARM registers at the beginning of larger basic blocks, which would eliminate the need for multiple memory accesses. In particular, this could be combined with optimizations such as *superblocks* [8] to achieve high performance emulation with the DBTs.

Acknowledgments

Arancini depends on Intel XED for translating input x86 machine code to IRancini and the Keystone Assembler for the ARM64 backend. It also uses the LLVM library extensively for its static backend and runtime components.

Many of the design decisions for the DBT implementation were taken together with Simon Kammermeier and Tom Spink, while Redha Gouicem provided both considerable expertise and very helpful advice. Likewise, special thanks to Pramod Bhatotia for the Guided Research opportunity.

Availability

The source code for Arancini is open-source and available on GitHub.

References

[1] musl libc. `https://musl.libc.org/`, 2005-2021. Accessed: April 17, 2023.

[2] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGOPS Oper. Syst. Rev.*, 40(5):2–13, oct 2006.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[4] Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. Performance analysis and predictability of the software layer in dynamic binary translators/optimizers. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, New York, NY, USA, 2013. Association for Computing Machinery.

[5] Derek L. Bruening and Saman Amarasinghe. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* PhD thesis, USA, 2004. AAI0807735.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach,.* Morgan Kaufmann, Waltham, MA, 2012.

[7] R. N. Horspool and N. Marovac. An Approach to the Problem of Detranslation of Computer Programs. *The Computer Journal*, 23(3):223–229, 08 1980.

[8] Wen mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, 1993.

[9] Koh M. Nakagawa. Project Champollion: Reverse engineering Rosetta 2, 2021.

[10] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, 1999.

[11] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[12] Bor-Yeh Shen, Jyun-Yan You, Wuu Yang, and Wei-Chung Hsu. An llvm-based hybrid binary translation system. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 229–236, 2012.

[13] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes.* The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.