Efficient Code Generation in a Region-based Dynamic Binary Translator

Efficient Code Generation in a Region-Based Dynamic Binary Translator

Tom Spink Harry Wagstaff Björn Franke Nigel Topham

Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh

t.spink@sms.ed.ac.uk, h.wagstaff@sms.ed.ac.uk, bfranke@inf.ed.ac.uk, npt@inf.ed.ac.uk

Abstract

Region-based JIT compilation operates on translation units comprising multiple basic blocks and, possibly cyclic or conditional, control flow between these. It promises to reconcile aggressive code optimisation and low compilation latency in performance-critical dynamic binary translators. Whilst various region selection schemes and isolated code optimisation techniques have been investigated it remains unclear how to best exploit such regions for efficient code generation. Complex interactions with indirect branch tables and translation caches can have adverse effects on performance if not considered carefully. In this paper we present a complete code generation strategy for a region-based dynamic binary translator, which exploits branch type and control flow profiling information to improve code quality for the common case. We demonstrate that using our code generation strategy a competitive region-based dynamic compiler can be built on top of the LLVM JIT compilation framework. For the ARM V5T target ISA and SPEC CPU 2006 benchmarks we achieve execution rates of, on average, 867 MIPS and up to 1323 MIPS on a standard X86 host machine, outperforming state-of-the-art QEMU-ARM by delivering a speedup of 264%.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Incremental Compilers

General Terms Design, experimentation, measurement, performance

Keywords Dynamic binary translation; region-based just-in-time compilation; alias analysis

1. Introduction

Dynamic binary translation (DBT) is a widely used technology that makes it possible to run code compiled for a target platform on a host platform with a different instruction set architecture (ISA). With DBT, machine instructions of a program for the target platform are translated to machine instructions for the host platform during the execution of the program. Among the main uses of DBT are *cross-platform virtualisation* for the migration of legacy applications to different hardware platforms (e.g. APPLE ROSETTA and IBM POWERVM LX86 (Stahl and Anand 2010) both based on TRANSITIVE's QUICKTRANSIT, or HP ARIES (Zheng and Thompson 2000)) and the provision of *virtual platforms* for convenient software development for embedded systems (e.g. VIRTUAL PROTOTYPE by SYNOPSYS).

Efficient DBT heavily relies on Just-in-Time (JIT) compilation for the translation of target machine instructions to host machine instructions. Although JIT compiled code generally runs much faster than interpreted code, JIT compilation incurs an additional overhead. For this reason, only the most frequently executed code fragments are translated to native code whereas less frequently executed code is still interpreted. Of central concern are the *size* and *shape* of these translation units presented to the JIT compiler: While smaller code fragments such as individual instructions or basic blocks take less time for JIT compilation, larger fragments such as linear traces or regions comprising control flow offer more scope for aggressive code optimisation (Aycock 2003). For this reason, many modern DBT systems rely on regions as translation units for JIT compilation and several different region selection schemes have been proposed in the literature (Bruening and Duesterwald 2000; Hiniker et al. 2005; Hiser et al. 2006b; Hsu et al. 2013). However, it remains an open question as how to efficiently exploit such regions for JIT code generation resulting in improved performance.

Our main contribution is a *complete, region-based* JIT *code generation strategy* considering optimal handling of branch type information and region exits, registration of JIT compiled code in translation caches, continuous profiling and recompilation, region chaining, and host code generation including custom alias analysis. The **key ideas** can be summarised as follows: We collect branch type information during code discovery and profiling and only expose region entries and indirect branch targets, whereas direct branch targets are neither accessible from outside the region nor through the indirect branch target table. This directly improves code quality as unnecessarily exposed branch targets defeat control and data flow analysis. Only identified region entries are registered in the translation cache, we do not allow arbitrary entry to a region. Again, this optimisation aids control and data flow analysis and, thus, ultimately improves performance. In addition, we provide shortcuts for region exits and implement region chaining, improving the transition from one region to another. We continuously profile execution, grow and recompile regions using up-to-date profiling information to include newly discovered blocks and transitions. Finally, we apply a custom alias analysis for host code generation, exploiting knowledge about the structure of the code, which is difficult to uncover using standard alias analysis. Whilst some of these techniques have been investigated before in isolation, we combine them, for the first time, to a complete region-based JIT compilation strategy inside a DBT system.

We have implemented our novel code generation strategy in our multi-threaded DBT system targeting the ARM V5T ISA on a standard 12-core `x86-64` host machine. We demonstrate its effectiveness across the SPEC CPU2006 integer benchmarks, where our system achieves an average execution rate of 867 MIPS, and up to 1323 MIPS. This is about 2.64 times faster than state-of-the-art QEMU-ARM.

1.1 Motivating Example

Most DBT systems will use some form of CPU state structure that contains the active state of the register file and any CPU flags – along with other control information. A DBT that works on an instruction-by-instruction basis will usually access this structure for

Listing 1. Example ARM assembly

```
BEGIN: mov r2, #1
LOOP:  cmp r0, 0
       beq END
       mul r2, r2, r0
       sub r0, r0, #1
       b LOOP
END:   mov r0, r2
```

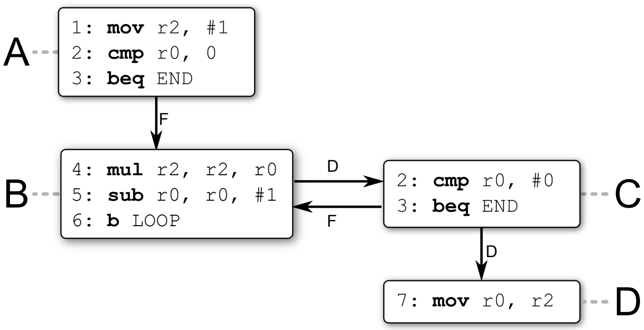

Figure 1. Blocks discovered by the profiler for the example code in Listing 1. An edge labelled **F** denotes a fall-through from a predicated branch, and an edge labelled **D** denotes a direct branch target.

every target instruction being executed, as most instructions will involve a read or write to one or more registers. However, a DBT that translates on a block-by-block basis (such as (Bellard 2005)) will typically assume that executing a basic block is an atomic operation, and can introduce optimisations that only update the CPU state structure once the entire basic block has been executed. This is because intermediate values from the results of target instructions can be kept in host registers, and re-used throughout the block until the last moment. This important optimisation significantly reduces the amount of reads and writes to memory, and can therefore greatly increase performance.

Traditional region-based DBTs work on a block-by-block basis, and will allow entry to the region via any block that is part of the region, however the consequence of this is that the address of each basic block must be taken, and doing so prevents any kind of *inter-block* optimisation. Whilst *intra-block* optimisations can still be applied, more aggressive *inter-block* optimisations cannot, as guarantees about CPU state must be maintained on entry to each block. In contrast, trace-based DBTs generate inherently linear control-flow graphs, which are only ever entered from the top (the *trace head*) and are usually only exited from the bottom. This enables optimisations to be applied across the entire trace but due to the lack of *interesting* control-flow, they miss out on certain loop optimisations.

The benefit of a region-based DBT is that non-linear control-flow is allowed within the region, which can lead to optimisations that would not be possible with linear control-flow (e.g. loop optimisations), but this benefit is restricted if addresses of individual blocks within the region are taken and, e.g. inserted to an indirect branch target table. This limits the ability of the optimiser to keep intermediate values (such as loop induction variables) in host registers, and to defer updating the CPU state structure until an exit point is reached.

The code given in Listing 1 (and the subsequent *control flow graph* (CFG) given in Figure 1), shows a simple ARM function that calculates the factorial of a number, supplied in `r0`. If native code was to be generated for this sequence, and we allowed entry to the sequence via any block, then each basic block would need to load the values of the registers in use from the register file, and cannot re-use values from a predecessor. Furthermore, at the end of a basic block, the register file must be updated with any changes in register values. This particular problem can pollute native code with unnecessary loads and stores when certain blocks are not actually

Listing 2. Native code with block addresses taken

```
BLOCK_A:
  movl $1, 8(%edi)
  movl 0(%edi), %eax
  test %eax, %eax
  jz BLOCK_D
BLOCK_B:
  movl 8(%edi), %ecx
  movl 0(%edi), %eax
  imul %eax, %ecx
  movl %ecx, 8(%edi)
  subl $1, %eax
  movl %eax, 0(%edi)
BLOCK_C:
  movl 0(%edi), %eax
  test %eax, %eax
  jnz BLOCK_B
BLOCK_D:
  movl 8(%edi), %eax
  movl %eax, 0(%edi)
```

Listing 3. Native code without block addresses taken

```
BLOCK_A:
  movl $1, %ecx
  movl 0(%edi), %eax
  test %eax, %eax
  jz END
LOOP:
  mul %eax, %ecx
  subl $1, %eax
  jnz LOOP
END:
  movl %ecx, 0(%edi)
  movl %ecx, 8(%edi)
```

Figure 2. Host machine code generated using a naïve scheme and using our integrated, region-based code generation methodology.

region entries, and with careful profiling and capturing of CFG edge information, it can be determined which blocks are internal to the region.

In the example CFG, block *A* is a region entry, and blocks *B*, *C* and *D* are only branched to by control-flow from other blocks. Two branch fall-through edges exist as $\overrightarrow{AB}$ and $\overrightarrow{CB}$, and two direct branches exist as $\overrightarrow{BC}$ and $\overrightarrow{CD}$. It is important to note that there are two basic blocks (*A* and *C*) discovered with overlapping code. This is because (given the input $r_0 \geq 1$) the profiler will discover the fall-through edge $\overrightarrow{AB}$ first, and then discover the direct edge $\overrightarrow{BC}$ that branches inside *A*, and hence creates a new basic block *C* containing the latter half of *A*.

If entry was allowed via any block, target register values would need to be loaded from the CPU state structure in each block – ensuring that the correct register values are used. This would be detrimental in performance, especially in the case of the loop between *B* and *C*, as the value of the induction variable in `r0` would need to be read from memory in *C* and written to memory in *B*, rather than keeping `r0` in a host register.

However, if we change the constraints to only allow entry via block *A*, and keep *B*, *C* and *D* as *region local* blocks, then we can produce an optimised form that loads initial register values into a host CPU register, which is reused throughout the loop, until we exit the code sequence and require that the updated register values are written back in to the CPU state structure.

This difference is clearly demonstrated in Listing 2 and Listing 3, where Listing 2 shows an example of x86 assembly generated for the code sequence described in Listing 1. When every block has its address taken, the block must access memory to request the value of the target machine register from the state structure. In Listing 3, we can see that an optimised form can be generated where host registers are used to track the state of the target machine register, until the very end where the values are written back to memory. This removes all memory accesses from the loop between block *B* and *C*, and can exploit host ISA features to generate an extremely efficient loop.

In general, our guiding principle is speculation and optimisation for the common case, i.e. we use profiling information on branch types, region entries, and indirect branch targets immediately for code optimisation even if there is the possibility of later updates of this information, possibly initiating re-compilation.

1.2 Contributions

This paper is not concerned with developing new ways of region selection, but its focus is on a strategy for efficient code generation and optimisation for regions once these have been formed using any of the techniques presented in the literature (Bruening and

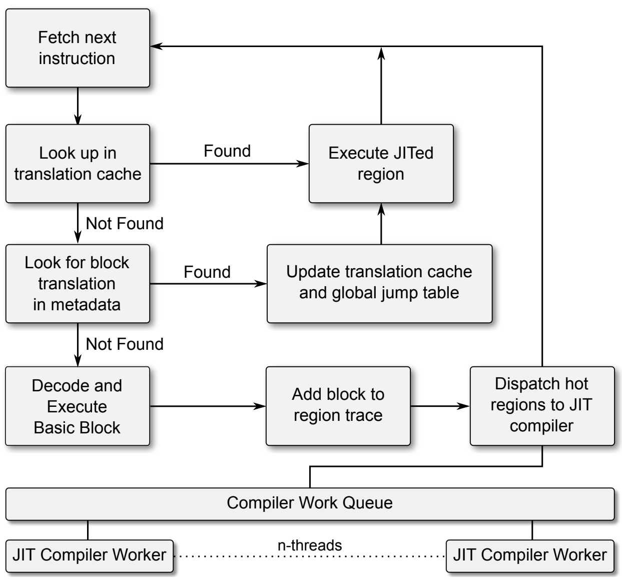

Figure 3. Main execution loop of our retargetable DBT system with decoupled, concurrent JIT compilation threads.

Duesterwald 2000; Hiniker et al. 2005; Hiser et al. 2006b; Hsu et al. 2013). Neither do we propose another technique for resolving indirect branches, but we show how branch type and control flow information can be exploited on top of any of the existing mechanisms for resolving indirect branches (Hiser et al. 2007; Dhanasekaran and Hazelwood 2011; Koju et al. 2012; Yin et al. 2012; Jia et al. 2013). Overall, we make the following contributions in this paper:

1. We introduce a **complete, region-based JIT code generation strategy** suitable for integration in high-speed DBT systems,
2. we demonstrate how to exploit **branch type profiling information** to enable improved back-end code generation including **loop optimisation**,
3. we introduce light-weight **region chaining**, borrowing concepts from trace chaining, and
4. we develop a new **custom alias analysis** that allows us to more accurately separate independent memory accesses, again enabling improved back-end code generation.

1.3 Overview

The remainder of this paper is structured as follows. In Section 2 we provide the background to our DBT system and, in particular, the region selection scheme used throughout this paper. This is followed by the presentation of our novel code generation strategy in Section 3. We present our empirical evaluation in Section 4 before we discuss related work in Section 5. Finally, we summarise and conclude in Section 6.

2. Background

2.1 DBT System Overview

Figure 3 shows the main execution loop of our DBT, which employs an interpretive component and farm of concurrent JIT compiler threads to achieve maximum speed. We initially begin by running the target program through the interpreter and collect profiling information about the basic blocks by building a region-oriented *control flow graph* (CFG). Once a region has been determined to exceed a certain threshold it will be dispatched to a JIT compiler worker, which will translate the region to native code. This process is asynchronous, and the target program will continue executing in the interpreter. Once the native code has been compiled, it will be made available by registering *region entry points* in block metadata and when the interpreter encounters a registered block, it will update the translation cache and begin executing the native code.

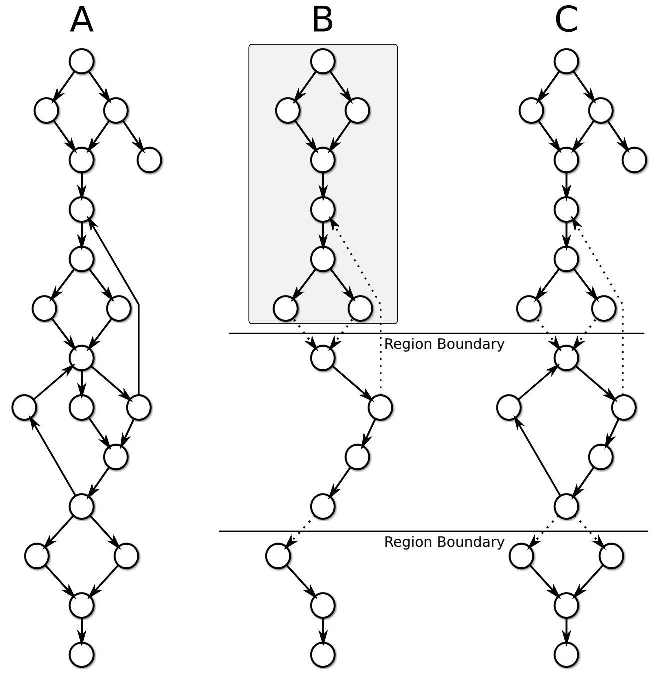

Figure 4. (A) Example of a whole-program control flow graph. (B) Parts of the control flow graph from (A) dynamically discovered after some time of execution, including forced region limits at page boundaries. (C) Additional control flow has been dynamically discovered after some more time executing the program.

Once inside native code, execution will remain there as long as blocks are available to execute. If a block is encountered that has not yet been compiled, control will return to the interpreter and profiling information updated accordingly. Gathering further profiling information about a region may lead to a region becoming eligible for recompilation, which gives rise to progressively optimal code, much like *tiered* or *staged* compilation (Joshi et al. 2004).

2.2 Region Selection

In a DBT system, region selection is concerned with forming the *shape* of translation units, where a region is typically a collection of basic blocks connected by control flow edges. This stage follows code discovery and profiling and it determines the boundaries of a fragment of recently discovered target code, and prepares it for translation into native host code. A number of region selection schemes for use in JIT compilers and DBT systems have been developed, e.g. (Bruening and Duesterwald 2000; Hiniker et al. 2005; Hiser et al. 2006b; Hsu et al. 2013). The focus of these papers has been on *policies* for region selection, i.e. decisions on how far and for how long to grow a region, but they do not explore code generation strategies for regions. Often regions are distinguished from *traces*, whilst technically traces are degenerate regions they are often treated separately due to their linear shape, i.e. the absence of multiple control flow successors and, in particular, loops.

JIT compilers present in e.g. JAVA VMs would have meta-information about the structure of the program being executed, and could use this information for *method*-based region selection techniques. But, the presence of meta-information is not guaranteed and cannot be relied upon, and indeed is not present in a raw instruction stream, so the DBT must rely on dynamic profiling information to effectively perform region selection (Whaley 2001). In this paper we use a page based region selection scheme similar to the one presented in (Böhm et al. 2011). Such a scheme enables efficient MMU emulation and detection of self-modifying code through page protection mechanisms provided by the OS. As shown in Figure 4(B) we start building a dynamic CFG and insert basic blocks and control flow edges between wherever we encounter dynamic control flow. After a certain interval (in terms of blocks executed in the interpreter) we scan the CFG and form regions, depending on the temperature and whether this is above a certain, adaptive threshold. In our scheme page boundaries are also compulsory region boundaries. Regions are then passed to the JIT compiler for code generation, and profiling execution continues, possibly extending the dynamically discovered CFG further (see Figure 4(C)).

3. Methodology

3.1 Overview

Our DBT begins executing a target program by means of an interpreter, which collects profiling information about execution flow as it executes. The interpreter executes basic blocks of instructions at a time, and edge information is collected about these blocks. Metadata structures, which describe regions, are used to track the "temperature" of a region, and when a "hot" region is detected, a *translation work unit* is dispatched to a compiler work queue. An idle JIT compiler worker thread picks up this work unit, and begins compilation. A work unit consists of a list of basic blocks to compile (which represents blocks within one region), the associated control-flow graph connecting those basic blocks together and a list of the blocks which are *region entries*. The compiler then translates each block in turn (on an instruction-by-instruction basis) into LLVM IR. Finally, when each block has been compiled, a *local jump table* (sometimes also referred to as *indirect branch target buffer*) is generated, which contains the addresses of each block that is a *region entry* block and each block that is the target of an indirect jump.

The *region prologue* is a small piece of set-up code common to each *region function*, which loads values that are reused throughout the native code (such as pointers to the various CPU state structures). Following this setup, an indirect branch via the previously generated *local jump table* is performed to begin execution at the desired basic block. A *region function* therefore, contains the translated native code for every block discovered (and marked as hot) in the region, and invoking this function will branch to the block that is to be executed, by accessing the *program counter* from the CPU state structure.

It is important to note that not all basic blocks that have been compiled have their addresses taken and corresponding entries registered in the *local jump table*. This constraint means that non-*region-entry* basic blocks cannot be entered from outside the region. The consequence, and indeed benefit, of not taking addresses of certain basic blocks allows LLVM to be more aggressive during the optimisation, phase – potentially merging basic blocks together and performing inter-block optimisations.

In Figure 4, the control-flow graph labelled *A* describes the *actual* control-flow of the target program, where *B* and *C* show the *discovered* control-flow, along with region boundaries. The shaded portion of *B* is magnified in Figure 5, which shows how blocks within a region are compiled to a region function, and how the function chains to other region functions by means of the *global jump table*.

3.2 Translation Lookup Cache

The *translation lookup cache* is a structure that lives in the execution engine component of the DBT and is used to resolve addresses of basic blocks to native code. In fact, it is a mapping of block addresses to the region function that contains the native translation of a particular block. Only *region entry* blocks are entered in to the translation cache, as it is only possible to branch to *region entry* blocks from the *local jump table*.

3.3 Region Chaining

Chaining is becoming a common feature in trace based JIT compilation systems, such as in the DALVIK VM and TRACEMONKEY. This technique typically involves profiling execution flow between compiled traces, and updating the translated code for hot edge source nodes of inter-trace jumps, to jump directly to the destination translation unit. We extend *trace chaining* to *region chaining*, which deals with hot control flow between regions. This can be the result of hot inter-region edges emerging only after some warmup time, where region selection has already partitioned code into regions, or due to unavoidable region limits such as page boundaries introduced by the region selection scheme (see also Section 2.2).

To simplify code generation we implement a weak form of region chaining, where we keep a *global jump table* of translated regions. It is important to distinguish this from the *translation cache* – the *global jump table* is only a jump table at region/page granularity and is not used when transitioning from the interpreter into native code. Conversely, the *translation cache* contains translation information at basic block granularity and is only used when transitioning from interpreter to native code.

The global jump table contains one entry (initially empty) for each possible region. Each entry consists of a single function pointer. In our case, we have at most one region per page, so the jump table contains $4GB/8KB = 524,288$ entries. These entries are updated when a miss occurs in the *translation cache* described above. Since, when we retranslate a region, we invalidate the translation cache entry for that region, this ensures that the *global jump table* always points to the most up to date translation for each region.

The *global jump table* is used when it is determined that a translated branch might have another region as its destination. This determination is made differently depending on the circumstances:

1. For a direct branch: if the target is outside the current region, then the global jump table is used if the branch is taken.
2. For an indirect branch if no targets within the current region have been encountered so far: the global jump table is used immediately.
3. For an indirect branch, if one or more targets within the current region have been encountered: if the branch resolves to an address within the current region, then the *local jump table* is used, otherwise the *global jump table* is used.

Since the global jump table is initialised with 'empty' entries, the requested entry must be checked before it is used (essentially a null-pointer check). If the requested entry is empty, execution flow leaves translated code.

3.4 Branching

A basic block is defined as a single-entry, single-exit linear code sequence, and as such the *terminating instruction* is always a branch to one or more basic blocks. There are two types of branches that can be made out of a basic block:

- **Direct:** A branch whose destination is known at JIT compilation time, i.e. the destination is a PC-relative or absolute address.
- **Indirect:** A branch whose destination is not known at JIT compilation time, i.e. a branch that uses a register value to calculate the destination.

These two cases can further be classified in to *predicated* and *non-predicated*, which impose additional constraints on the control-flow out of a basic block. When a branch is *predicated*, the fall-through block for the *branch not taken* case can be treated as a *direct* branch.

In Figure 5, each node in the CFG (except for *E*) has been discovered by the profiler, and as such the CFG has been compiled to LLVM IR on a block-by-block basis. Node *E* and the corresponding edge $\overrightarrow{CE}$ have not yet been discovered by the profiler, i.e. they have not yet executed, or have not exceeded the compilation threshold.

Nodes *A* and *F* are *region entries*, and *H* and *I* are the targets of indirect branches. As such, these nodes have their block addresses taken, and a corresponding entry added to the *local jump table*. The other nodes are never accessed by an indirect jump (as far as the current profiling information is concerned) so their block addresses are not taken, and no entry is registered in the *local jump table*.

This leads to the case where native code may be available for a basic block, i.e. it has been compiled, but it is not reachable from outside the region.

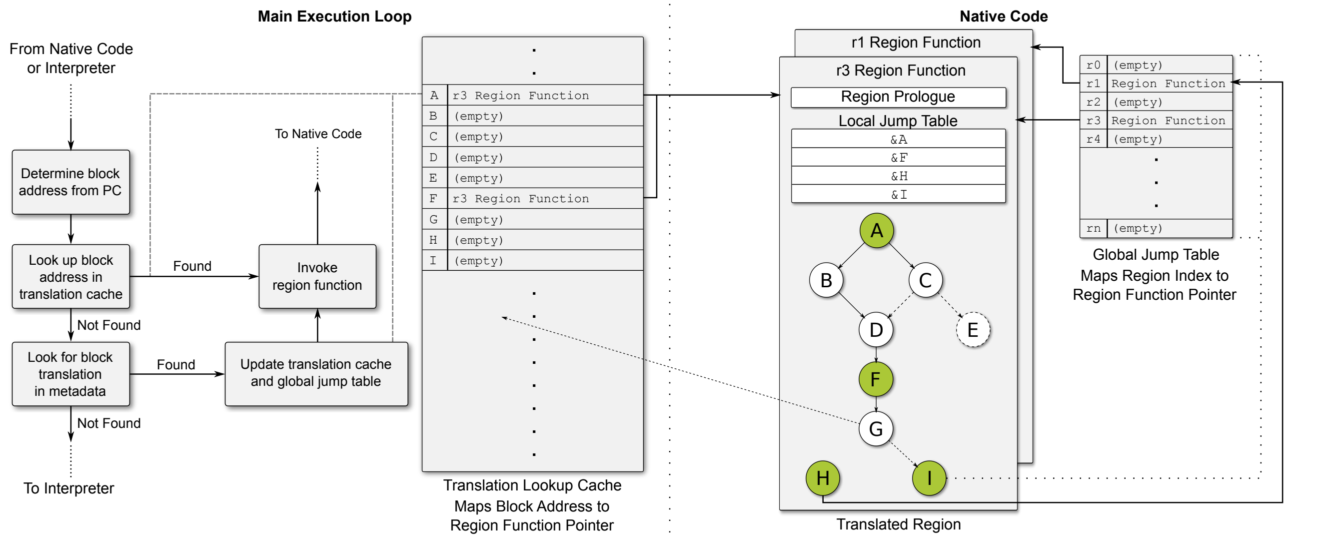

Figure 5. Interaction between regions via the *global jump table* and the internal interactions between basic blocks, either directly or via the *local jump table*. The control flow graph represents the region in the shaded area in Figure 4(B).

3.4.1 Direct Branches

Where we have a direct branch from basic block A to B, (and B has no indirect branch predecessors), we do not have to add the address of B to the local jump table and instead we can emit LLVM IR to perform a direct branch to B.

There are two approaches that we take when generating the proper control-transfer sequence, and they depend on whether or not the terminating branch is predicated or non-predicated.

For a non-predicated branch, given we know at compile time the jump target, if the target lies outside the region boundary, we generate code to transfer control via the *global jump table* – as shown by node H. This means we chain directly to the region containing the destination block (if available). If the target lies within the region, as shown by node A, then we can check to see if we are compiling that particular block in this work unit, and if so, we can emit LLVM IR that directly branches to it. If the destination block is not in the work unit, then we must return immediately to the interpreter, as a native translation is not available in this round of compilation.

For a predicated branch, the same sequence applies as before, except we first determine whether or not the branch is to be taken. If the branch is not taken, then the fall-through block is directly branched to (if present in the work unit).

3.4.2 Indirect Branches

As we cannot know at JIT compile time what the destination of an indirect branch might be, we have to rely on profiling information to assist in making decisions about how to transfer control from a basic block to a successor. An important point to note is that we can treat a predicated indirect branch instruction as having a single direct edge to the fall-through block, and treat this as in the direct branch case (Section 3.4.1).

If the edge information we receive at compile time contains no edges, then we must transfer control via the *global jump table*. This is demonstrated in Figure 5 as node I, and is because we know that the *local jump table* cannot satisfy our jump (since an entry would only be available if we have encountered that particular edge). Exiting via the *global jump table* is required because the indirect branch may be to a different region. If it turns out that this speculation is incorrect (or if the destination region does not contain a translation for the target block) we return to the interpreter.

If the edge information contains exactly one edge, then we can emit a simple comparison instruction to determine whether or not that edge should be taken. If the edge is correct, we branch directly to that basic block and otherwise fall back to the *global jump table*. Node C (before discovery of E) is an example of this, where we have a single indirect edge $\overrightarrow{CD}$, but have not yet discovered $\overrightarrow{CE}$.

Finally, for a block with multiple indirect successors (such as node G), we emit code to check that the target block lies within the same region, and if so we perform an indirect branch via the *local jump table*. If the target block lies outside the current region, we branch via the *global jump table*.

Other implementations of *local jump tables* are possible, e.g. some of the techniques presented in (Hiser et al. 2007; Koju et al. 2012; Jia et al. 2013; Yin et al. 2012; Dhanasekaran and Hazelwood 2011) could act as drop-in replacements, however, we have found our implementation to provide sufficiently low lookup times and high hit rates.

3.5 Region Registration in Translation Caches

Every basic block that is encountered by our DBT has metadata held about it, which describes certain properties about the block, and contains a pointer to the region function containing its implementation, if it has been identified as a region entry. When the execution engine begins executing a block, it looks up the block metadata and checks to see if a native translation exists – if so, the translation cache is updated and native code is entered. Additionally, the *global jump table* is updated with a pointer to the function for the region containing the block. If a region is recompiled, the block metadata will be updated to reflect the new function pointer and the change would propagate through to the translation cache.

3.6 Continuous Profiling and Recompilation

The mixing of instructions and data, and the presence of indirect branching make it impossible to fully and accurately determine the precise control flow of a program from machine code only. Although techniques exist which attempt to extract control flow information from programs statically (Kinder et al. 2009) these often must be extremely conservative and thus DBT systems using them suffer from poor performance.

On the other hand, techniques for extracting control flow information at run time are becoming increasingly effective (Joshi et al. 2004). These techniques often do not capture all possible control flow paths through a program in their first pass – thus, it is necessary to profile continuously.

We may therefore discover new control flow within regions which we have already translated and compiled. If we do not retranslate the relevant regions when we encounter such control flow at run time we can only evaluate it sub-optimally. For example, we may discover that a block which we previously excluded from the region *local jump table* is in fact a region entry. In this case, we must return to the interpreter to execute this block, since we do not have a translation entry for it.

Our technique does not require any special treatment for the retranslation of regions. Instead, the profiling system does not distinguish between already translated and non-translated regions. If previously-untranslated code or control flow is encountered in a translated region, it is executed using the interpreter and profiled. If it is frequently executed and becomes hot, the full region will be retranslated in order to include the new code and control flow.

3.7 Host Machine Code Generation

A *translation work unit* is the unit provided to a JIT compiler worker thread and consists of a list of *basic block descriptors*, along with basic block edge information, representing a particular region. The *basic block descriptors* contain a list of decoded instructions. Each instruction in a block is translated to LLVM IR one-by-one, using a technique similar to (Wagstaff et al. 2013) and once the instructions have been translated, a *block epilogue* is emitted. This epilogue is generated based on the type of control-flow associated with the block, and essentially contains the IR that transfers control to the next block.

Finally, after all the blocks in the *translation work unit* have been compiled, and the region prologue has been generated, a single LLVM function remains that represents the region just compiled. This function is then passed through the LLVM optimiser, as described in Section 3.7.1.

After the optimisation passes have completed, the LLVM IR is compiled to native machine code using the LLVM JIT compiler interface and when the native code is available, each basic block that is marked as a *region entry* has a pointer to the newly compiled function stored in its metadata.

3.7.1 LLVM Optimisation Passes

During the translation phase, an LLVM module is built containing the function that represents the region being translated. The module also contains helper functions, which are highly amenable to inlining. All the helper functions are marked as internalisable, and an inlining pass is applied. Typically, the helper functions will provide a very small function (such as reading the PC register, or writing to target machine memory), and are easily inlined.

After inlining, the resulting module is subjected to a number of LLVM passes, based on the standard CLANG `-O3` optimisation level. The main difference is that instead of using an LLVM provided alias analysis implementation, we use ours as described in Section 3.7.2.

Since we allowed some basic blocks not to be region entry points, this has opened up more scope for aggressive loop optimisation, which yields the full benefit of a region-based JIT. With a trace-based JIT, loop optimisations rarely happen, as traces are inherently linear. However, with our region-based approach, we can perform a significant amount of loop optimisations across the control-flow within a region, which would also not be possible if we allowed entry to the region from any basic block.

3.7.2 Alias Analysis

Alias analysis of pointers is an important phase that enables further program optimisations to reason better about data flow. For example, a *dead store elimination* pass uses pointer aliasing information to determine whether or not a redundant store to a memory location can be eliminated, based on any memory accesses that happen between those stores.

Listing 4 shows how incomplete pointer aliasing information can lead to the optimiser being unable to remove dead stores. The stores on lines 1 and 5 are killed by the store on line 7, but because the optimiser cannot detect that the operations on pointers in lines 2-4 do not alias, it cannot remove the stores. This directly translates to machine code as shown in Listing 5, which is safe (and correct), but in our case not at all optimal.

Listing 4. LLVM IR after dead store elimination

```
1 store i32 36076, i32* %4
2 %42 = load i64* inttoptr (i64 61931224 to i64*)
3 %43 = add i64 %42, 6
4 store i64 %43, i64* inttoptr (i64 61931224 to i64*)
5 store i32 36076, i32* %4
6 ...
7 store i32 36092, i32* %4
```

Listing 5. X86 machine code after target lowering

```
movl $37076, 60(%r12)
addq $6, 61931224
movl $37076, 60(%r12)
...
movl $36092, 60(%r12)
```

Figure 6. Remaining dead-stores in LLVM IR after optimisation, and resulting X86 machine code due to incomplete alias analysis.

Vendor & Model	DELL™ POWEREDGE™ R610
Number cores	2×6
Processor Type	2× Intel© Xeon™ X5660
Clock/FSB Frequency	2.80/1.33 GHz
L1-Cache	$2 \times 6 \times$ 32K Instruction/Data
L2-Cache	$2 \times 6 \times$ 256K
L3-Cache	2× 12 MB
Memory	36 GB across 6 channels
Operating System	Linux version 2.6.32 (x86-64)

Table 1. DBT Host Configuration.

In the example shown in Figure 6, the problem stems from the alias analysis implementation (quite correctly) being unable to determine whether or not the pointer held in `%4` aliases with the constant pointer value `61931224`. Assuming that `%4` and `61931224` alias is a safe assumption and as such generates safe code. But, armed with the knowledge about the working of our DBT, we know that `%4` contains a pointer to a CPU state register, and that the constant pointer is an address that does not intersect with the CPU state structure, hence we can say that they do not alias. Providing this guarantee to LLVM's *dead store elimination* optimisation pass enables the pass to remove the redundant stores, and generate better code. The particular example described above is important for region-based compilation, as redundant updates to the CPU state are eliminated, hence reducing the number of memory operations occurring in a particular sequence.

When a loop is involved, keeping target machine register values in host registers instead of constantly reading and writing to the CPU state structure improves performance significantly – but this kind of loop optimisation can only work to its full potential when combined with the *jump table optimisation* technique described in Section 3.4.

4. Experimental Evaluation

4.1 Experimental Methodology

We have evaluated our DBT code generation approach using the SPEC CPU2006 integer benchmark. It is widely used and considered to be representative of a broad spectrum of application domains. We used it together with its *reference* data sets. The benchmarks have been compiled using the GCC 4.6.0 C/C++ cross-compilers, targeting the ARM V5T architecture (without hardware floating-point support) and with `-O2` optimisation settings.

We have measured the elapsed real time between invocation and termination of each benchmark in our DBT system using the UNIX `time` command on the host machine described in Table 1 with our DBT system configured as in Table 2. We used the average elapsed wall clock time across 10 runs for each benchmark and configuration in order to calculate execution rates (using MIPS in terms of target instructions) and speedups. For summary figures we report harmonic means weighted by dynamic target instruction

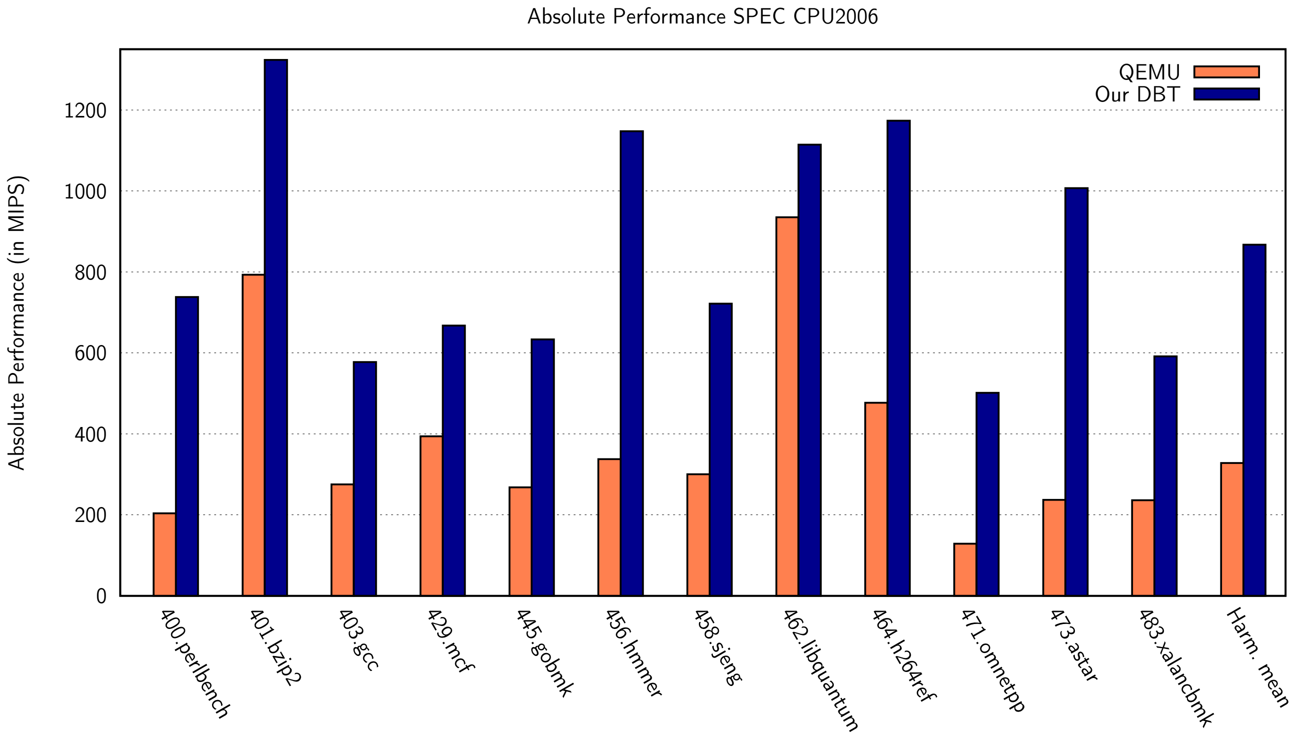

Figure 7. Absolute performance figures (in MIPS) for the *long-running* SPEC CPU2006 integer benchmarks for both QEMU-ARM and our DBT, indicating that the quality of the generated code by our system is superior to the code generated by QEMU-ARM.

DBT Parameter	Setting
Target architecture	ARM V5T
Host architecture	X86-64
Translation/Execution Model	Asynch. Mixed-Mode
Tracing Scheme	Region-based (Böhm et al. 2011)
Tracing Interval	30000 blocks
Translation Cache	8192 Entries
JIT compiler	LLVM 3.4
No. of JIT Compilation Threads	10
JIT Optimisation	`-O3` & Part. Eval. (Wagstaff et al. 2013)
Initial JIT Threshold	20
Dynamic JIT Threshold	Adaptive (Böhm et al. 2011)
System Calls	Emulation
Floating Point	Software Emulation (‘`soft`’)

Table 2. DBT System Configuration.

count. For the comparison to the state-of-the-art we use the ARM port of QEMU 1.4.2 as a baseline.

Additionally, we have also evaluated our DBT using the EEMBC-1.1 benchmark suite. These benchmarks are typically shorter running and serve to evaluate the performance of the JIT compiler portion of our DBT. In order to normalise performance to particular duration, we adjusted the iteration count of each benchmark so that it ran for about ten seconds in QEMU, then we invoked the benchmark with the same iteration count in our DBT and measured performance in the same manner as for SPEC.

4.2 Experimental Results for SPEC CPU2006

Figure 7 gives an overview of the absolute performance of QEMU vs. our DBT. In every case, we improve on QEMU, and on average achieve a 2.64*x* improvement in absolute performance.

The biggest improvement is achieved for 473.astar, which can be attributed to the benchmark responding well to our ability to apply loop optimisations within a region. The relative performance improvement of 473.astar when *region chaining* is enabled is negligible, and so indicates that the majority of time is spent in region local code. Aggressive loop optimisations are performed within this region (where the bulk of the algorithm lies). This explains the excellent performance improvement over QEMU, which performs no such optimisations. This explanation can also be applied to 464.h264ref, which benefits greatly from our ability to optimise loops better than QEMU.

The smallest improvement is for 462.libquantum, which may be due to the benchmark itself being heavy in arithmetic instructions, but not so much in looping constructs. This particular characteristic explains the excellent performance of QEMU, and hence why we only see a 1.2*x* improvement in this case. QEMU's block-based optimisations work well here, due to the linear nature of the arithmetic instructions and larger basic block sizes.

Interestingly, the relative performance improvements as optimisations are enabled (shown in Figure 8) of 462.libquantum are similar to that of 473.astar, and the absolute performance of both the benchmarks are within the same area - but 462.libquantum is already fast in QEMU.

4.3 Impact of Optimisations

Figure 8 shows how combinations of the optimisations described in Section 3 affect the relative performance of the DBT. The baseline is using standard LLVM `-O3` optimisation and partial evaluation, but without any of our optimisations described in the paper applied.

Overall, the addition of our custom alias analysis improves every benchmark, except for 429.mcf. On average this gives a 1.32*x* performance improvement, but it is the combination of all our strategies that yield the best result. *Jump table optimisation* on its own does not give rise to a significant performance improvement, but responds well when combined with alias analysis. This may be due to the fact that the most interesting optimisation to apply across basic blocks is to remove dead stores and to keep host registers live with frequently used values (potentially from the CPU state structure). Without the precise aliasing information this kind of optimisation is not possible to do effectively, and so the combination of both *jump table optimisation* and *custom alias analysis* give rise to the best performance improvements.

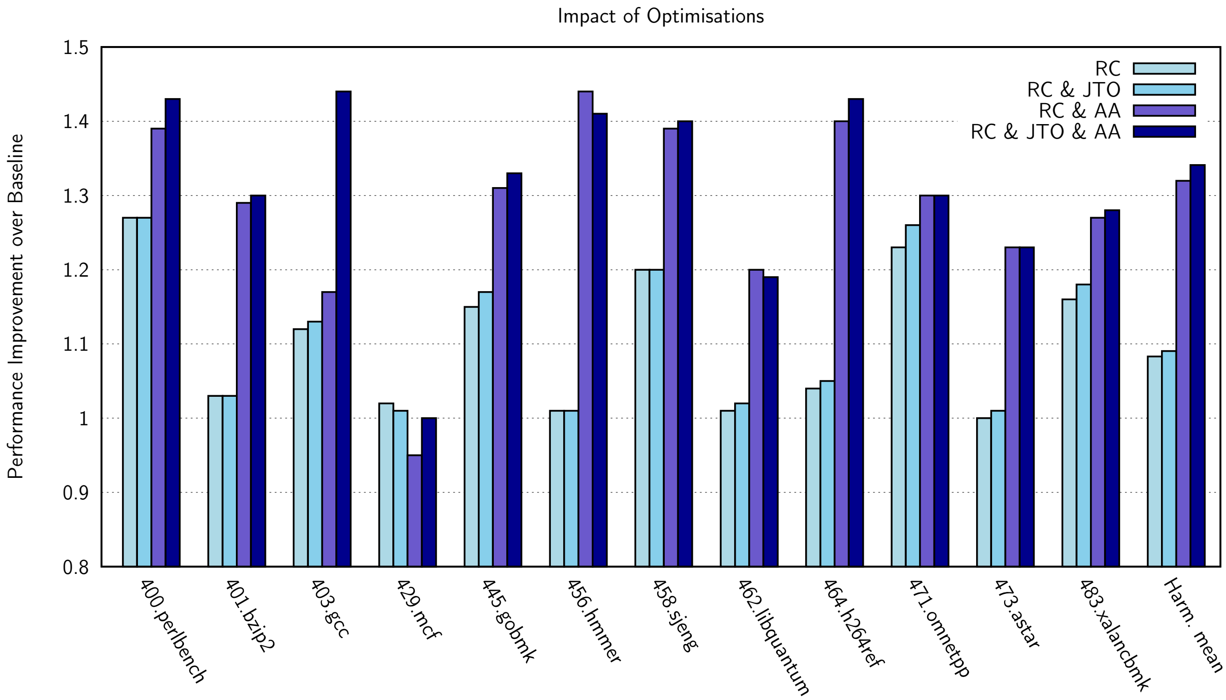

Figure 8. Breakdown of performance impact of different optimisations. Baseline is standard LLVM -O3 and partial evaluation (Wagstaff et al. 2013) at JIT compilation time. Additional region chaining (RC), jump table optimisation (JTO) and alias analysis (AA) complement each other.

473.astar remains at baseline performance when the *region chaining* optimisation is applied, and this may be due to the majority of execution being spent in region-local code. It has an absolute performance figure of > 1000 MIPS, which indicates fast running code, but the benefits of *region chaining* are minimal, due to the lack of inter-region control-flow.

403.gcc is a particularly control-flow heavy benchmark, and responds well to the combination of all the optimisations together. Also of interest is the 429.mcf benchmark, which does not consistently improve in performance like the majority of the other benchmarks. Despite this, 429.mcf is more than 1.5 times faster in our DBT system than in QEMU.

4.4 JIT Compilation Performance

The execution time of the SPEC CPU2006 benchmarks with their reference data sets is dominated by the time spent executing native code, whereas the fraction accounted for JIT compilation time is small. For such long-running benchmarks *code quality* is paramount and this where our region based code optimisations outperform simpler basic block or trace based schemes. However, JIT compilation time is still important for shorter-running applications, or programs that exhibit phased behaviour and, hence, exercise the JIT compiler more heavily. To evaluate JIT compilation performance of our DBT system we have run additional, smaller benchmarks, where time for JIT compilation constitutes a larger portion of the overall time (see Figure 9). In every case, we beat QEMU in absolute execution performance, but as in the SPEC results, our relative performance improvements vary greatly. As can be seen, the most significant result here is that we execute fft00 at a rate of 6138 MIPS compared to QEMU's 3897.95. However, this only shows a modest relative performance gain of $1.5x$, where as idctrn01 outperforms QEMU by $2.85x$. We can attribute these variances again to the characteristics of individual benchmarks in the suite, where we can say that in the benchmarks which are amenable to loop optimisations, i.e. contain more intra-region loops, we show a greater *relative* performance improvement. Overall, these results demonstrate that even for shorter-running applications where JIT compilation latency plays a greater role than absolute code quality our system is highly competitive despite its use of larger translation units and aggressive code optimisations.

5. Related Work

5.1 Region based DBT Systems

Region based JIT compilation has been used for some time in JAVA virtual machines, e.g. (Suganuma et al. 2003, 2006), but has only been considered more recently for DBT systems (Jones and Topham 2009; Böhm et al. 2011; Kaufmann and Spallek 2013). The reason for this late adoption of region based policies has been presumably the increased latency for compilation and optimisation of larger regions, which has only been addressed recently with the introduction of decoupled, latency-hiding JIT task farms (Böhm et al. 2011). The bulk of the work in this field has focused on region selection, though, and less on code generation and optimisation for dynamically discovered regions. In (Jones and Topham 2009) large translations units, i.e. regions, are introduced for dynamic binary translation and region selection policies based on strongly connected components, control flow graph fragments and OS pages are compared. A refined page based region selection scheme is developed in (Böhm et al. 2011) and combined with a parallel JIT compilation task farm. Specific optimisations for a DBT system, which compiles target- to host code via JVM bytecode, are considered in (Kaufmann and Spallek 2013).

5.2 Code Generation and Optimisation in DBT Systems

Most DBT systems appear to have adopted a code generation strategy operating on individual basic blocks or linear traces of basic blocks. For example, QEMU uses such an approach using its own *tiny code generator* (TCG) and additional block chaining, translation caching and lazy condition evaluation (Bellard 2005). DYNAMO (Bala et al. 2000) is a dynamic optimisation system, i.e. the input is an executing native instruction stream. DYNAMO uses an interpreter for initial execution until a "hot" instruction sequence is identified. At that point, DYNAMO generates an optimised version of the trace into a software code cache. DYNAMO treats backward branches as trace delimiters, i.e. traces are by definition linear. After translation it emits an optimised single-entry, multi-exit,

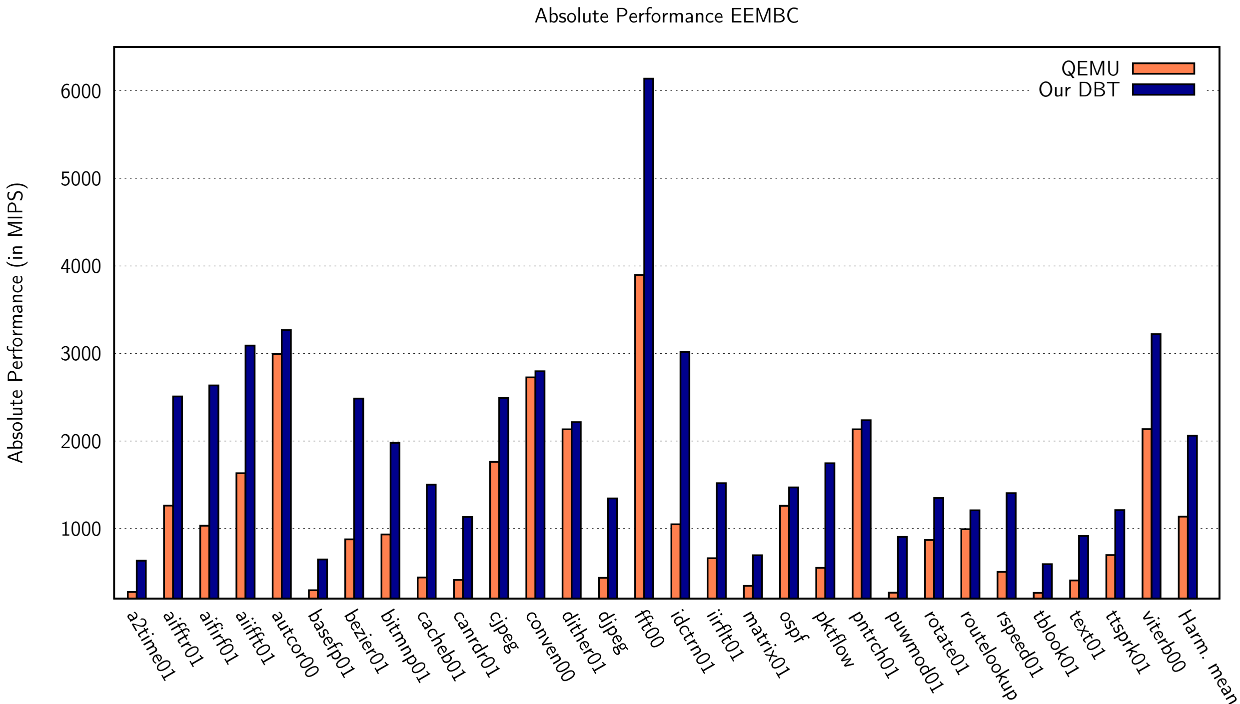

Figure 9. Absolute performance figures (in MIPS) for the *shorter-running* EEMBC benchmarks for both QEMU-ARM and our DBT, indicating that JIT startup time and compilation performance of our DBT is more than competitive with QEMU-ARM despite aggressive code optimisations applied by our system.

contiguous sequence of instructions for each trace. Trace optimisation in DYNAMO considers branch types, but is generally less aggressive than our scheme, which utilises additional loop optimisations. DYNAMORIO (Bruening et al. 2003) is a successor of DYNAMO. DYNAMORIO operates on two kinds of code sequences: basic blocks and traces. Both have linear control flow, with a single entrance and potentially multiple exits, but no internal join points. Optimisations are restricted to the linear control flow present in traces. The single-entry multiple-exit format simplifies analysis algorithms, but limits the scope of optimisations that can be applied. STRATA (Hiser et al. 2006a) is a retargetable DBT system offering additional uses for dynamic instrumentation and optimisation. Different fragment selection policies (Hiser et al. 2006b) have been evaluated for STRATA, however, all of these have in common that they are linear traces, possibly spanning branch or function call boundaries. STRATA uses chaining of traces to avoid overheads associated with returning to the main execution loop after every native trace. An ARM port of STRATA considers architecture-specific optimisations, e.g. relating to the exposed PC (Moore et al. 2009). The optimisations performed by UQDBT – a machine-adaptable dynamic binary translator – are discussed in (Cifuentes and Emmerik 2000; Ung and Cifuentes 2001). This tool uses an algorithm for finding hot paths using edge weight profiles, and optimises code in a machine-independent way, based on hot path information. Whilst units of translation in UQDBT are basic blocks, for its hot path (re)optimisation it groups hot basic blocks and their connecting control flow edges into regions. The paper focuses primarily on newly discovered hot paths and locality transformations, but does not provide a complete code generation strategy. A particular aspect of code generation in DBT systems, namely recovery of jump table case statements, is discussed in (Cifuentes and Emmerik 1999). Alias analysis for DBT systems is considered in (Guo et al. 2006), but unlike our approach this requires runtime checks.

5.3 DBT Systems Using LLVM for JIT Compilation

A parallel and concurrent JIT compilation task farm for use in DBT systems is presented in (Böhm et al. 2011). The JIT compiler is based on the LLVM framework, which is used for translation of paged regions of target instructions to host instructions. The paper discusses a particular region selection scheme and parallel JIT compilation, but provides no details of the actual code generation approach used. LNQ (Hsu et al. 2011) extends QEMU with an LLVM based JIT compiler, but does not consider code regions for translation. It uses linear traces instead. HQEMU (Hong et al. 2012) is a multi-threaded dynamic binary translator, which extends QEMU with multiple instances of the LLVM compiler for JIT compilation. Similar to our system HQEMU builds on top of LLVM, but it only operates on linear traces and does not support region-based compilation. Unfortunately, direct performance comparisons are hampered as the paper only reports relative improvements over an unusual baseline, which we were unable to verify or repeat.

6. Summary & Conclusions

In this paper we have developed a novel, integrated approach to JIT code generation within region-based DBT systems. We exploit branch type information, introduce region chaining, develop selective region registration in translation caches, add on continuous profiling and recompilation, and finally include custom alias analysis to enable aggressive code optimisations, which would not be possible in a JIT scheme based on linear traces. We demonstrate the efficiency of our region-based JIT code generation approach using the SPEC CPU2006 benchmarks compiled for the ARM V5T ISA, which our DBT system translates on-the-fly to the host machine's X86 ISA. In comparison to state-of-the-art QEMU-ARM we achieve an average speedup of 2.64, and up to 4.25 for individual benchmarks. We show that each of the techniques developed in this paper on their own contributes to increased code quality, but it is the particular combination of code generation steps that results in performance improvements greater than the sum of its parts.

References

J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857077. URL `http://doi.acm.org/10.1145/857076.857077`.

V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-

2. doi: 10.1145/349299.349303. URL http://doi.acm.org/10.1145/349299.349303.

F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1247360.1247401.

I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 74–85, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993508. URL http://doi.acm.org/10.1145/1993498.1993508.

D. Bruening and E. Duesterwald. Exploring optimal compilation unit shapes for an embedded just-in-time compiler. In *In Proceedings of the 2000 ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3*, pages 13–20, 2000.

D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL http://dl.acm.org/citation.cfm?id=776261.776290.

C. Cifuentes and M. V. Emmerik. Recovery of jump table case statements from binary code. In *Proceedings of the 7th International Workshop on Program Comprehension*, IWPC '99, pages 192–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0179-6. URL http://dl.acm.org/citation.cfm?id=520033.858247.

C. Cifuentes and M. V. Emmerik. UQBT: Adaptive binary translation at low cost. *IEEE Computer*, 33(3):60–66, 2000.

B. Dhanasekaran and K. Hazelwood. Improving indirect branch translation in dynamic binary translators. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering, and Virtualized Environments*, RESoLVE'11, pages 11–18, 2011.

B. Guo, Y. Wu, C. Wang, M. J. Bridges, G. Ottoni, N. Vachharajani, J. Chang, and D. I. August. Selective runtime memory disambiguation in a dynamic binary translator. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 65–79, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33050-X, 978-3-540-33050-9. doi: 10.1007/11688839_6. URL http://dx.doi.org/10.1007/11688839_6.

D. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2440-0. doi: 10.1109/MICRO.2005.22. URL http://dx.doi.org/10.1109/MICRO.2005.22.

J. D. Hiser, N. Kumar, M. Zhao, S. Zhou, B. R. Childers, J. W. Davidson, and M. L. Soffa. Techniques and tools for dynamic optimization. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 279–279, Washington, DC, USA, 2006a. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://dl.acm.org/citation.cfm?id=1898699.1898797.

J. D. Hiser, D. Williams, A. Filipi, J. W. Davidson, and B. R. Childers. Evaluating fragment construction policies for SDT systems. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 122–132, New York, NY, USA, 2006b. ACM. ISBN 1-59593-332-8. doi: 10.1145/1134760.1134778. URL http://doi.acm.org/10.1145/1134760.1134778.

J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: 10.1109/CGO.2007.10. URL http://dx.doi.org/10.1109/CGO.2007.10.

D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104–113, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259030. URL http://doi.acm.org/10.1145/2259016.2259030.

C.-C. Hsu, P. Liu, C.-M. Wang, J.-J. Wu, D.-Y. Hong, P.-C. Yew, and W.-C. Hsu. LnQ: Building high performance dynamic binary translators with existing compiler backends. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 226–234, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4510-3. doi: 10.1109/ICPP.2011.57. URL http://dx.doi.org/10.1109/ICPP.2011.57.

C.-C. Hsu, P. Liu, J.-J. Wu, P.-C. Yew, D.-Y. Hong, W.-C. Hsu, and C.-M. Wang. Improving dynamic binary optimization through early-exit guided code region formation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 23–32, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1266-0. doi: 10.1145/2451512.2451519. URL http://doi.acm.org/10.1145/2451512.2451519.

N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang. SPIRE: improving dynamic binary translation through SPC-indexed indirect branch redirecting. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 1–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1266-0. doi: 10.1145/2451512.2451516. URL http://doi.acm.org/10.1145/2451512.2451516.

D. Jones and N. Topham. High speed CPU simulation using LTU dynamic binary translation. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 50–64, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-540-92989-5. doi: 10.1007/978-3-540-92990-1_6. URL http://dx.doi.org/10.1007/978-3-540-92990-1_6.

R. Joshi, M. D. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 239–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://dl.acm.org/citation.cfm?id=977395.977660.

M. Kaufmann and R. G. Spallek. Superblock compilation and other optimization techniques for a Java-based DBT machine emulator. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, pages 33–40, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1266-0. doi: 10.1145/2451512.2451521. URL http://doi.acm.org/10.1145/2451512.2451521.

J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 214–228, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-540-93899-6. doi: 10.1007/978-3-540-93900-9_19. URL http://dx.doi.org/10.1007/978-3-540-93900-9_19.

T. Koju, X. Tong, A. I. Sheikh, M. Ohara, and T. Nakatani. Optimizing indirect branches in a system-level dynamic binary translator. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, pages 5:1–5:12, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1448-0. doi: 10.1145/2367589.2367599. URL http://doi.acm.org/10.1145/2367589.2367599.

R. W. Moore, J. A. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser. Addressing the challenges of DBT for the ARM architecture. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '09, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-356-3. doi: 10.1145/1542452.1542472. URL http://doi.acm.org/10.1145/1542452.1542472.

E. Stahl and M. Anand. A comparison of PowerVM and x86-based virtualization performance. Technical Report WP101574, IBM Techdocs White Papers, 2010.

T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 312–323, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: 10.1145/781131.781166. URL http://doi.acm.org/10.1145/781131.781166.

T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*, 28(1): 134–174, Jan. 2006. ISSN 0164-0925. doi: 10.1145/1111596.1111600. URL http://doi.acm.org/10.1145/1111596.1111600.

D. Ung and C. Cifuentes. Optimising hot paths in a dynamic binary translator. *SIGARCH Comput. Archit. News*, 29(1):55–65, Mar. 2001. ISSN 0163-5964. doi: 10.1145/373574.373590. URL http://doi.acm.org/10.1145/373574.373590.

H. Wagstaff, M. Gould, B. Franke, and N. Topham. Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *Proceedings of the Annual Design Automation Conference*, DAC '13, pages 21:1–21:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2071-9. doi: 10.1145/2463209.2488760. URL http://doi.acm.org/10.1145/2463209.2488760.

J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 166–179, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9. doi: 10.1145/504282.504295. URL http://doi.acm.org/10.1145/504282.504295.

L. Yin, J. Haitao, S. Guangzhong, J. Guojie, and C. Guoliang. Improve indirect branch prediction with private cache in dynamic binary translation. In *International Conference on High Performance Computing and Communication and International Conference on Embedded Software and Systems (HPCC-ICESS)*, pages 280–286, 2012. doi: 10.1109/HPCC.2012.45.

C. Zheng and C. Thompson. PA-RISC to IA-64: transparent execution, no recompilation. *Computer*, 33(3):47 –52, Mar. 2000.