Fast Binary Translation: Translation Efficiency and Runtime Efficiency

Mathias Payer
Department of Computer Science
ETH Zurich

Thomas Gross
Department of Computer Science
ETH Zurich

Abstract—**Fast binary translation is a key component for many modern software techniques. This paper reflects on the implementation of fastBT, a generator for low-overhead, table-based dynamic (just-in-time) binary translators. We discuss the most challenging sources of overhead, propose optimizations to further reduce these penalties, and present a detailed performance analysis with different approaches to translate indirect control transfers. To allow comparison with other systems the paper includes an empirical evaluation of fastBT relative to three other binary translation systems (DynamoRIO, HDTrans, and PIN).**

The fastBT generator is architecture-neutral but this paper's analysis and performance data focus on IA-32 Linux. fastBT performs well in practice: we report an overhead of 0% to 10% for the majority of benchmarks. fastBT uses a trace cache and trampolines to obtain efficiency in translation and execution of the translated program. The keys to fastBT's good performance are a configurable inlining mechanism optimizations for the different forms of indirect jumps.

To ease development of a binary translator, the translation actions of fastBT are specified in high-level abstractions that are compiled into fastBT's translation tables. This table generator allows a compact description of the transformations that the binary translator effects in the translated code.

I. Introduction

Binary translation (BT) may add, remove, or replace individual instructions of the translated program. There are two approaches to binary translation (BT): (i) static (ahead-of-time) translation before the execution of the program and (ii) dynamic (just-in-time) translation at runtime. Both approaches to BT may result in a slow-down of the translated program (relative to the original program), since the added or translated instructions might add overhead. The efficiency of the translated program has therefore received considerable attention by various researchers, as we discuss in more detail in Section V. The key advantage of static translation is that the translation process does not incur a runtime penalty. The principal disadvantage is that static BT is limited to code regions that can be identified at compile time. In many environments this approach is not appropriate: dynamically loaded libraries pose a problem, and the static translation cannot cope with self-modifying code. Dynamic BT translates code as it is executed. Using dynamic BT, the runtime system is able to adapt to phase changes of a program and to optimize hot code regions incrementally, and we therefore focus on dynamic translation. Most dynamic translators include a code cache to lower the overhead of translation. Translated code is placed in this cache, and subsequent executions of the same code region can benefit from the already translated code.

There are two key aspects in the design of a fast binary translator. First, the translation process must be lean; although this overhead is paid only once, it still impacts every program. Second, the overhead introduced through the translation must also be low; this overhead is paid for every execution of a given code region.

The biggest overhead for BT is caused by indirect control transfers since they result in an additional runtime lookup. Three kinds of indirect control transfers exist:

1) **Indirect jumps:** The target depends on a memory location or a register and is not known at translation time. Therefore the lookup happens at runtime.
2) **Indirect calls:** Similar to indirect jumps the target is not known at translation time and can change for subsequent calls.
3) **Function returns:** The target is on the stack. The stack of the user program always contains untranslated addresses because some programs depend on return addresses for features, e.g., exception management.

fastBT is a generator for low-overhead, low footprint, table-based dynamic binary translators that allows us to investigate the issues in constructing a binary translator that is fast and that produces fast code. fastBT brings many well-known concepts of BT together and allows their investigation in a stable context. BT is important, but a simple combination of existing techniques is far from trivial. The contribution of this paper is an investigation of the combination of a number of approaches to optimize indirect control transfers, the development of novel strategies to translate indirect control transfers (like the inlined fast return strategy or the return and indirect call prediction that use inlined caches of the last targets), and providing a high-level interface without sacrificing performance.

The current implementation provides tables for the Intel IA-32 architecture, and uses a per-thread trace cache for translated blocks. The output of the binary translator is constructed through user-defined actions that are called in the just-in-time translator and emit IA-32 instructions. The translation tables are generated from a high-level description and are linked to the binary translator at compile time. The user-defined actions and the high-level construction of the translation tables offer an adaptability and flexibility that is not reached by any other translator we know of. E.g., the fastBT approach allows in a few hundred lines of code the implementation of a library that dynamically detects and redirects all memory accesses inside a transaction to a software transactional memory system.

Like HDTrans [22], fastBT leaves the stack of the user program unchanged. Although we use the stack for fastBT's functions, the user program never sees a change of the original layout and is completely unaware of the translation. Based on this decision fastBT is able to (i) handle self-modifying code that changes its own return address, (ii) handle exceptions without additional overhead, (iii) simplify debugging of the application. Each one of these points accesses the program stack to match return addresses with known values. These comparisons would not work if the stack was changed, because the return addresses on the stack would point into the trace cache, instead of into the user program. Therefore fastBT replaces every return instruction with an indirect control transfer to return to translated code.

II. Design and implementation

To meet the two key requirements of a fast binary translator (lean translation, low-overhead for translated code) design and implementation must go together, and these two cannot be separated.

The translator processes blocks of the original program, places them in the trace cache and adds entries to the mapping table. The execution flow of the program stays in the trace cache. If an untranslated basic block is executed, then it is translated on the fly, and the new trace is added to the trace cache. See Figure 1 for an overview.

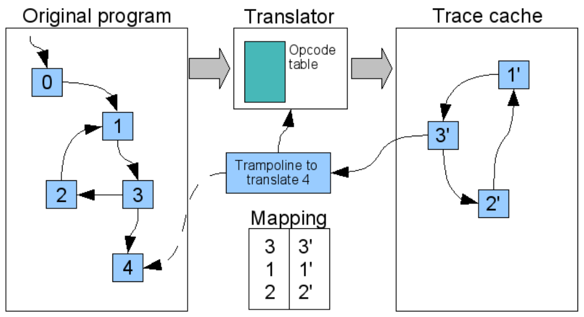

Fig. 1. Runtime layout of the binary translator.

A. Table generation

The BT library uses a *multilevel* translator table with information about each possible machine code instruction. Writing translation tables by hand is hard and error prone, as there exist a multitude of different possible instructions and combinations with prefixes on IA-32, and accounting for the various encodings and formats of all machine instructions can be a cumbersome task.

fastBT uses a table generator that offers a high-level interface to all instructions instead, see Figure 2. The programmer can specify how to handle specific instructions or where to insert or to remove code. This attractive feature allows a compact description of changes in the translated code with a single point of change.

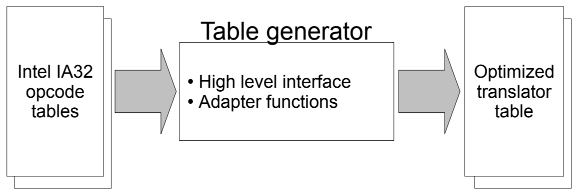

Fig. 2. Compile time construction of the translator table.

The supplied base tables contain detailed information about register usage, memory accesses, different instruction types (e.g., if the FPU is used) and information about individual instructions. These base tables define an identity transformation where only control flow instructions are altered to keep the execution flow under the control of the translator.

Using adapter functions, the user can specify which types or instructions are altered when the table is produced. The table contains an action for each existing machine code instruction. These actions can be selected during table generation based on the instruction, the properties of the instruction (e.g., register access, memory access, control flow, function calls, and so on). Figure 3 shows an adapter function that redirects all instructions that might access memory to a special handler. The output of the table generator is then compiled and linked together with the fastBT sources to the binary translator library, which is then used as a shared library in the program.

```
bool isMemOp (const unsigned char* opcode,
 const instr& disInf, std::string& action)
{
  bool res;
  /* check for memory access in instr. */
  res = mayOpAccessMem(disInf.dstFlags);
  res |= mayOpAccessMem(disInf.srcFlags);
  res |= mayOpAccessMem(disInf.auxFlags);

  /* change the default action */
  if (res) { action = "handleMemOp"; }

  return res;
}

// in main function:
addAnalysFunction(isMemOp);
```

Fig. 3. Adapter function that redirects all instructions that access memory to a special action - C++ Code.

B. Basic translator

The layout of the translation engine is a simple table-based iterator. When the translator is invoked with a pointer to a basic block, it first allocates a new buffer fragment in the trace cache. The translator then adds a reference from the original basic block to the buffer fragment into the mapping table. Next the basic block gets translated one instruction at a time.

The previously generated tables are used to decode each instruction. Because IA-32 instructions can differ in length (1 byte to 16 bytes) fastBT uses a multilevel table: If the decoding of the instruction is not finished in the current table, a pointer redirects to another table, and the translator reads the next byte. This process is repeated until the instruction is decoded completely and only possible arguments are left.

In the next step the instruction and the arguments are passed to the corresponding *action function*, which handles the instruction. The action functions are *completely unrestricted* in how they refactor the code. They can *add arbitrary code* (e.g., function calls) and *alter the instruction* in any conceivable way or *remove the instruction* altogether.

Translation stops at recognizable basic block (BB) boundaries like branches and return instructions. Some BB boundaries are not recognizable; if a branch target points into an already translated block, a part of this block is translated again.

At the end of a basic block the translator checks if the targets of outgoing edges are already translated, and adds jumps to already translated targets. Otherwise the translator builds a trampoline that translates the target, if it is executed, and adds a jump to this trampoline.

C. Predefined actions

The translator needs different action functions to support the identity transformation. The simplest action functions do nothing, thereby removing the instruction in the instrumentation, or copy the instruction verbatim to the cache. Additional action functions are needed to keep the execution flow inside the cache.

One action function handles direct unconditional jumps and redirects them to another block in the cache. Other action functions handle conditional jumps and redirect the execution flow to the translated targets. Special action functions handle indirect jumps, indirect calls and return instructions. They need an indirect control transfer that cannot be determined at translation time, and these functions add runtime checks to transfer control to the correct destination.

All action functions that handle direct control transfers check if the target is already translated. If it is not translated, they will generate a jump to a trampoline that translates and executes the target if the trampoline is called.

Simple action functions copy the instructions to the code buffer, without altering them. Additional actions are needed to redirect the execution flow so that it stays inside the buffer. These actions ensure that the program cannot escape the translation and that no unchecked or untranslated instructions are executed.

D. Trace cache

During the execution of a program it is likely that certain code regions are executed multiple times. Hence keeping translated code in a cache potentially reduces the overhead of BT since the translated code regions can be reused an arbitrary number of times without generating additional overhead.

As reported in [4] and [7] code sharing between threads is low for desktop applications and moderate for server applications. Therefore fastBT uses a *per thread cache*-strategy to increase code locality and to potentially extract better traces.

An additional important advantage of thread local caches is that the translator can directly emit hard-coded pointers to thread local data structures into the cache without the need to call expensive lookup functions to determine thread local data structures. This feature is used in the translator itself for *syscall* and *signal* handling and to inline and optimize all thread local accesses from the instrumented program.

The combination of fastBT's basic translator and the trace cache leads to a greedy trace extraction. Traces are formed as a side effect of the first execution and placed in the trace cache.

E. Trampoline table

Every time when the translator finishes a basic block, it must emit continuation points. To keep the trace cache clean from these translation hints the translator generates a trampoline and emits a jump to the trampoline in the cache (`jmp trampoline_addr`). The trampoline contains all the information needed to start the translation at the continuation point, the location where the trampoline was called, and information about the thread.

As soon as the trampoline is used and the location it points to is translated the translator can backpatch the jump to the trampoline with a jump to the translated block.

F. Mapping table

The mapping hash table contains entries for all basic blocks. An entry in this table consists of a pointer into the user program and a pointer into the trace cache. This decision leads to an 8 byte stride for this table.

The hash function is designed to return a relative offset in the hash table. Adding the result from the hash function to the base pointer of the mapping table returns the address of a table entry. Inlined machine code can check entries efficiently this way.

The hash function needs to be fast, flexible and inlineable. The overhead for hashing should be low, and therefore a complex hashing mechanism was avoided. After tests with different SPEC CPU2006 benchmarks and other binaries fastBT uses the simple hash function `(addr<<1) & (HASH_PATTERN)`. The original address is shifted one bit to the left and binary anded with `HASH_TABLE_SIZE - 1` and the last three bits are set to 0. In this implementation the hash size must always be a power of two.

In the case of a hash collision (e.g., the address value is not equal `NULL`) the entry is stored in the next empty slot.

G. Signal and syscall handling

Instrumented environments need special treatment for signals and system calls. A task or thread can schedule individual signal handler functions and execute system calls. The kernel transfers control from kernel-space back to user-space after a system call or after the delivery of a signal. A user-space binary translator cannot control these control transfers, but must rewrite any calls that install signals or execute system calls.

fastBT catches signal handlers and system calls and wraps them into trampolines that return the control flow to translated code. Currently fastBT is able to translate signals installed by *signal* and *sigaction* and all system calls, covering both interrupts and the *sysenter* instruction. The *sysenter*-handling is of course specific to the syscall handling of the Linux kernel 2.6 [12].

H. Overheads in translated code

In this section we discuss the runtime overhead of the translated program. The overhead introduced into the translated program is the major source of any slowdown relative to the unmodified binary; the constant translation and initialization overheads are negligible in most cases.

Indirect jumps, indirect calls and return instructions: These instructions are the most frequent instructions that incur a runtime overhead.

In all three cases the target of the execution is unknown and the program must look up a pointer to the original program in the mapping table to get the pointer into the trace cache. The translation scheme is similar in all cases as stated in Figure 4. As an indirect call is only a push of the current `eip` and an indirect jump, the translator simply needs to add the push. Return instructions are also similar to indirect jumps. This instruction pops a value from the stack and indirectly jumps to this value.

`jmp reg/mem`	`call reg/mem`	`ret`
–	`push eip`	–
`push reg/mem`	`push reg/mem`	(addr. on stack)
`push tld`	`push tld`	`push tld`
`call ind_jmp`	`call ind_jmp`	`call ind_jmp`

Fig. 4. Translation of indirect jumps/calls and return instructions, *tld* is thread local data.

If the target is already translated and the lookup in the mapping table is a hit in the first row, then the call of `ind_jmp` adds 21 instructions. This overhead is very high and various optimizations (described in Section III) try to reduce this overhead. If the target is not translated, then the translator must translate this region first. Even more overhead results from hash misses where the lookup function loops through the mapping table to find the correct entry.

In these cases a single instruction in the original code is mapped to multiple instructions in the translated code block. An indirect jump adds 20 instructions in the best case if the fast mapping table lookup (described in section III-A) is used, and even more instructions are used to recover from a miss.

I. Statistics

To evaluate optimizations and to get some statistics about the programs, fastBT offers profiling information that can be activated to count important numbers like: number of translated instructions and basic blocks, number of executed indirect jumps, number of function calls, and others.

III. Optimization

Function calls and the following return appear as matching pairs in the execution stream, and this relationship offers promising optimization points. The occurence of indirect jumps in general and the lookup in the mapping table offers another opportunity.

Profiling shows that most overhead is generated by indirect jumps and some overhead is generated by the mapping table lookup. The overhead for the initialization and translation is negligible if not all used pages are initialized with, e.g., the *halt* instruction (this can be very useful for debugging purposes).

A. Fast mapping table lookup

The second largest overhead results from the mapping table lookup. Due to the design of the hash function, fastBT has a low collision rate of nearly 0% for most benchmark programs (see Table II).

To benefit from this fact fastBT implements a fast mapping table lookup in assembly code that is inlined into the trace cache. This code sequence only checks the first element; if this location is not a direct hit, then the lookup falls back to the slower function with a loop to retrieve the target address.

B. Return optimizations

Return instructions represent a major fraction of indirect jumps that cause the largest instrumentation overhead for fastBT. fastBT offers several different optimization strategies for return instructions. These optimizations reduce the costs of the indirect control transfers.

Shadow return stack: To reduce the number of lookups and indirect jumps, fastBT implements a shadow return stack similar to FX!32 [9]. This stack contains only pairs of return addresses and translated addresses.

Call instructions are extended by some additional instructions that push the translated address and the current instruction pointer to the shadow stack.

Return instructions are then replaced by a check if the top of the stack is equal to the top of the shadow stack. If this check succeeds, then fastBT transfers control to the translated address on the shadow stack.

Although this optimization needs only 18 instructions to replace a call/return pair, and this code sequence is faster than the unoptimized version, it is in every case slower than the inlined fast return.

Return prediction: Return prediction leaves the original call unchanged, but caches the target of the last translated and untranslated return inline. When the code fragment with the return prediction is executed, the code first checks if the cached address matches the current address. A match causes a direct jump to the cached translated address, whereas a mismatch overwrites the cached address with the new target and branches to the translated address.

Figure 5 shows the code fragment for the inlined prediction. fastBT only executes 4 instructions, if the prediction is correct. An incorrect prediction needs 8 additional instructions next to an indirect jump with at least 35 instructions to fix the cached values leading to 43 instructions in total. To benefit from this optimization fastBT needs more than 48.7% correct predictions to outperform the indirect jump.

```
cmpl $cached_rip, (%esp)
je hit_ret
pushl tld
call ret_fixup
hit_ret:
addl $4, %esp
jmp translated_rip
```

Fig. 5. Return instruction with an included prediction, *rip* is the return instruction pointer and *tld* the pointer to thread local data.

Inlined fast return: This optimization inlines a fast mapping table lookup where needed into the trace cache. If the first lookup is a hit, then fastBT can branch directly to the translated target. Otherwise an indirect jump is used.

The HASH_PATTERN used in Figure 6 is already statically shifted right to remove the shift operation. fastBT also uses extended address calculation to reduce the number of instructions needed to get the hash table offset.

```
pushl %ebx
pushl %ecx
movl 8(%esp), %ebx # load rip
movl %ebx, %ecx
andl HASH_PATTERN, %ebx
subl maptbl_start(0,%ebx,4), %ecx
jecxz hit
popl %ecx
popl %ebx
pushl tld
call ind_jmp
hit:
movl maptbl_start+4(0,%ebx,4), %ebx
movl %ebx, 8(%esp) # overwrite rip
rip popl %ecx
popl %ebx
ret
```

Fig. 6. A translated return instruction with the fast return optimization, *rip* is the return instruction pointer and *tld* the pointer to thread local data.

Using this implementation, fastBT needs 12 instructions for a return in the best case with a direct mapping table hit.

C. Indirect call prediction

fastBT applies the concept of return address prediction to predict indirect calls as well. This optimization caches call targets, and if the prediction is correct execution can jump directly to the translated target. Otherwise fastBT calls a fixup routine that stores the new target in the cache and then jumps to the translated address.

This optimization is well suited for shared libraries as targets are loaded dynamically after the program has started and the library loader uses indirect calls. Furthermore the targets of these calls remain constant during program execution; as a consequence the target cache has a high hit rate.

We applied this optimization to *all* indirect jumps, but the result was a high miss rate with diminishing returns.

D. Function inlining

Profiling shows that function calls are responsible for the largest overheads. As fastBT already extracts traces into the trace cache, the addition of function inlining was an obvious extension.

This optimization served as a test of the extensibility of the fastBT framework. We implemented basic function inlining in no more than about 50 lines of code. Every time the translator sees a function call it checks the length of the callee, and if the function contains only one basic block it is inlined into the current basic block.

We also explored different strategies for inlining depth and aggressiveness to allow inlining of multiple functions and basic blocks, with diminishing returns for more complicated schemes.

IV. Performance evaluation

In the evaluation of fastBT we are interested in the comparative performance to other systems as well as in the instrumentation overhead. We would also like to know the significance of different optimizations for the individual benchmarks. Such performance data makes it possible to characterize the different benchmarks and to reason which other optimizations can be used for further improvements.

A. Experimental setup

All benchmarks were executed on a single machine with an Intel Core2 Duo CPU at 3.00GHz and 2GB main memory. The system runs on Ubuntu 7.10 with gcc version 4.1.3 and glibc version 2.6-22.

The SPEC CPU2006 benchmark suite version 1.0.1 is used to evaluate the single threaded performance. We compare fastBT with other binary translators: HDTrans [22] version 0.4.1, DynamoRIO [5; 6] version 0.9.4 and PIN [17] revision 19012.

B. Instrumentation overhead

Instrumentation overhead is measured compared to an execution without binary translation. fastBT is compiled with gcc and -O3. These fastBT optimizations are activated in various runs:

- Inlined fast return (IFR)
- Return prediction (RP)
- Fast mapping table lookup (FM)
- Prediction for indirect calls (PIC)
- Function inlining (FI)
- Fast indirect jump (FIJ)

Figure 7 shows the performance of fastBT compared to untranslated code. We measure the overhead factor of all C and C++ based SPEC CPU2006 benchmarks.

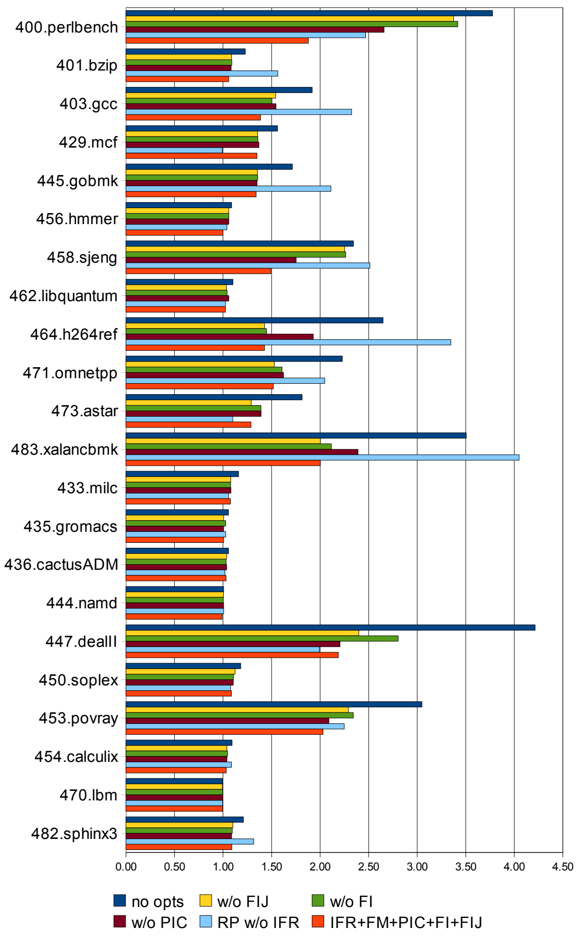

Fig. 7. SPEC CPU2006 benchmarks slowdown relative to untranslated code, comparing different optimizations

The fastest setting is a combination of an inlined fast return, the fast mapping table lookup, prediction for indirect calls, function inlining, and fast indirect jump (IFR+FM+PIC+FI+FIJ). All other bars refer to this implementation and replace or remove certain optimizations.

Performance is good for most benchmarks (0-10% overhead). A second group of benchmarks has a moderate overhead below 52%. A third group of five programs suffers from a high overhead ranging from 88% to 103%.

Overhead analysis: Table I shows characteristics of the different SPEC CPU2006 benchmarks. This table reveals that the gcc benchmark pays a high translation cost since the binary is invoked 9 times (and therefore the binary translator is also invoked 9 times). This initial translation cost is hard to amortize; the gcc benchmark is the only one where translation overhead is significant. Two other benchmarks, perlbench and gobmk, also have a relatively large code base and are started multiple times.

Table II presents a per benchmark statistic of the SPEC CPU2006 benchmarks using the fastest configuration (IFR+FM+PIC+FI+FIJ). This table, compared with Figure 7, explains why binary translation is in some cases slower than the untranslated run.

Benchmark	Instrs.	BBs	Execs.	Ovhd.
400.perlbench	**248,753**	**54,288**	**3**	1.88
401.bzip	72,261	11,809	6	1.06
403.gcc	**2,244,313**	**500,030**	**9**	1.39
429.mcf	8,207	1,467	1	0.99
445.gobmk	**540,224**	**99,735**	**5**	1.34
456.hmmer	19,667	3,523	1	1.00
458.sjeng	16,604	3,337	1	1.49
462.libquant.	8,209	1,230	1	1.03
464.h264ref	161,107	22,052	3	1.43
471.omnetpp	50,398	10,281	1	1.52
473.astar	27,755	4,698	2	1.10
483.xalan.	158,461	27,210	1	2.00
433.milc	20,237	2,750	1	1.06
435.gromacs	41,025	5190	1	1.01
436.cactus.	44,365	7,371	1	1.02
444.namd	36,545	3,790	1	0.99
447.dealII	103,520	13,784	1	1.99
450.soplex	79,152	11,606	2	1.08
453.povray	64,054	11,368	1	2.03
454.calculix	92,054	13,964	1	1.03
470.lbm	6,912	1,099	1	0.99
482.sphinx3	31,403	5,308	1	1.09

TABLE I
PER BENCHMARK STATISTIC WITH SUM OF TRANSLATED INSTRUCTIONS FOR ALL EXECUTIONS (ACTIVE CODE), NUMBER OF BASIC BLOCKS, HOW MANY TIMES THE BINARY TRANSLATOR WAS STARTED PER BENCHMARK RUN AND THE MINIMAL OVERHEAD/SLOWDOWN.

Most of the overhead results from the following four different sources:

- **Function pointers:** Calling a function through a function pointer results in an indirect call. Every indirect call leads to the execution of overhead instructions. The memory location of the function pointer is read and the target is then mapped into the trace cache. To reduce this overhead the last target of the function pointer is saved if the indirect call prediction optimization is activated. If the function pointer changes, then fastBT must do an additional lookup for the new address. The perlbench, gobmk, and sjeng benchmarks have a high collision rate in the mapping table.
- **Number of function calls:** Every function call incurs a linear overhead. The cost of function calls can be decreased through inlining. The dealII benchmark shows nicely that the overhead can be reduced through inlining; fastBT does not pay the additional cost for a function call in 54% of the calls.
- **Jump tables (switch):** Switch constructs or jump tables are another source of overhead. The C compiler translates switch constructs into indirect jumps. The targets of the memory locations do not change, but the binary translator must add and execute an additional lookup for each of these jumps.
- **Mapping cache misses:** A high miss rate in the mapping table leads to additional overhead for every indirect jump and every indirect call. This problem arises if the mapping table is over full or if multiple targets are close together.

Benchmark	Map. misses	(miss%)	Function calls	(inlined)	Ind. jumps	Ind. calls	(miss%)
400.perlbench	246,667	(0.00%)	**21,908,972,069**	(9.50%)	**21,929,721,015**	**3,902,298,779**	(**89.14%**)
401.bzip2	6	(0.00%)	6,686,411,394	(0.00%)	1,870,766	1,867	(9.86%)
403.gcc	41,172,650	(0.81%)	**11,416,485,263**	(2.82%)	**5,040,160,816**	653,553,951	(4.08%)
429.mcf	0	(0.00%)	6,937,298,559	(0.05%)	1,708,666	574,634	(0.01%)
445.gobmk	**23,892,158**	(**16.23%**)	**17,818,856,119**	(1.33%)	117,474,377	**185,811,452**	(**15.96%**)
456.hmmer	15	(0.00%)	219,205,278	(26.78%)	163,062,857	1,138,876	(0.01%)
458.sjeng	1	(0.00%)	**21,939,941,742**	(1.25%)	**10,992,904,415**	**5,070,023,325**	(**64.05%**)
462.libquant.	0	(0.00%)	1,762,122,564	(0.00%)	979	209	(7.18%)
464.h264ref	**1,133,513,780**	(**42.64%**)	9,148,416,877	(30.36%)	2,316,733,272	**28,445,058,103**	(1.20%)
471.omnetpp	**335,404,293**	(**10.57%**)	**17,282,090,091**	(19.29%)	3,151,849,446	2,733,835,044	(0.76%)
473.astar	100	(0.00%)	**17,389,970,212**	(31.63%)	10,809,621	4,996,462,097	(0.00%)
483.xalan.	51,717,068	(1.96%)	**19,825,915,425**	(13.87%)	2,426,718,169	9,161,983,117	(2.30%)
433.milc	0	(0.00%)	6,707,912,535	(1.43%)	11,999,314	3,856,839	(0.00%)
435.gromacs	2	(0.00%)	3,510,490,537	(75.48%)	27,444,253	3,274,839	(0.86%)
436.cactus.	5	(0.00%)	1,670,570,147	(0.53%)	1,650,753,737	223,565	(22.09%)
444.namd	2	(0.00%)	33,516,882	(20.47%)	14,909,541	1,969,176	(0.00%)
447.dealII	1,137,176	(0.01%)	**52,965,608,272**	(54.00%)	**21,163,171,969**	540,898,547	(0.13%)
450.soplex	235,035	(0.01%)	2,482,695,709	(3.54%)	1,722,689,963	27,488,899	(0.00%)
453.povray	49,590	(0.00%)	**18,698,952,109**	(8.10%)	433,067,223	**7,072,491,358**	(**29.10%**)
454.calculix	1,038	(0.00%)	5,200,465,040	(25.43%)	502,727,282	11,226,880	(0.00%)
470.lbm	0	(0.00%)	5,273,650	(49.98%)	2,627,289	3,724	(0.75%)
482.sphinx3	5,662	(0.00%)	7,489,332,386	(10.54%)	268,757,951	6,126,858	(0.06%)

TABLE II
PER BENCHMARK NUMBER OF MAPPING TABLE MISSES, PERCENTAGE COMPARED TO OVERALL MISSES, NUMBER OF CALLS, PERCENTAGE OF CALLS THAT ARE INLINED, NUMBER OF INDIRECT JUMPS, NUMBER OF (PREDICTED) INDIRECT CALLS AND PERCENTAGE OF MISPREDICTIONS.

This miss rate can be lowered with adaptive optimization through remapping of hot entries, a larger mapping table, or a different hash function. Each of these possible remedies leads to either additional overhead or a larger memory footprint and initialization cost.

Each of the four worst performing benchmarks (perlbench, sjeng, dealII and povray) include multiple of the above mentioned factors. The most dominant overhead is the combined effect of the collisions in the mapping table and the overhead induced by the indirect call handling.

C. Single-threaded comparative performance

Figure 8 shows the performance of fastBT relative to other binary translation systems.

The comparison between fastBT, DynamoRIO and PIN shows that fastBT performs overall as well as intermediate-representation-based (IR-based) translation systems that use intermediate representations instead of tables, but without the complexity of such a system. Exceptions are the perlbench, sjeng and povray benchmarks where fastBT is slower due to the high miss rate of our indirect call prediction.

An important difference between fastBT, HDTrans, and DynamoRIO is that fastBT supports the new sysenter system call interface. DyanmoRIO does not support this interface and exits with an error message. HDTrans fails to maintain control after the return of any sysenter system call and the program continues untranslated. Both systems support only the older interrupt driven system call interface.

D. Code size

The code (of the translator) generated by fastBT is compact. The final library fits into 88k of code, including all selected features and optimizations. HDTrans, DynamoRIO and PIN all need larger runtime environments with 160k of code, 324k of code and 8,607k of code respectively. Disadvantages of large libraries are additional cache misses and pollution of the instruction cache. Due to the small memory footprint fastBT interfers less with the original code.

V. RELATED WORK

Binary translation (BT) is extensively covered in the literature, and numerous binary translators exist both from academia and industry. BT dates back to the 80ies with work on Smalltalk-80 [10] and a simulator for an IBM System/370 [18] where complete code blocks were translated.

In this section we survey some binary translators that are similar to fastBT either in the target application or in usability. There are two kinds of binary translators: the first group (e.g., HDTrans, Mojo [23] and JudoDBR [20]) consists of low-level, table-based binary translators, and the second group (e.g., DynamoRIO, PIN, and Valgrind [19]) consists of high-level, IR-based implementations.

The advantage of the IR-based approaches is that more sophisticated optimizations are possible, but the disadvantage is that the translator needs to compensate a higher runtime overhead. Other binary translators like the one from Kistler et al. [16] or Hiser et al. [14] try to speed up program code using profiling and adaptive optimizations to overcome the additional instrumentation cost. FX!32 [9] uses a dual approach. On one hand it uses *cross-platform* emulation for infrequently executed code and on the other hand employs profile generation and offline static recompilation of hot code to speed up overall execution. A hash map is used to map between the emulator and the translated code.

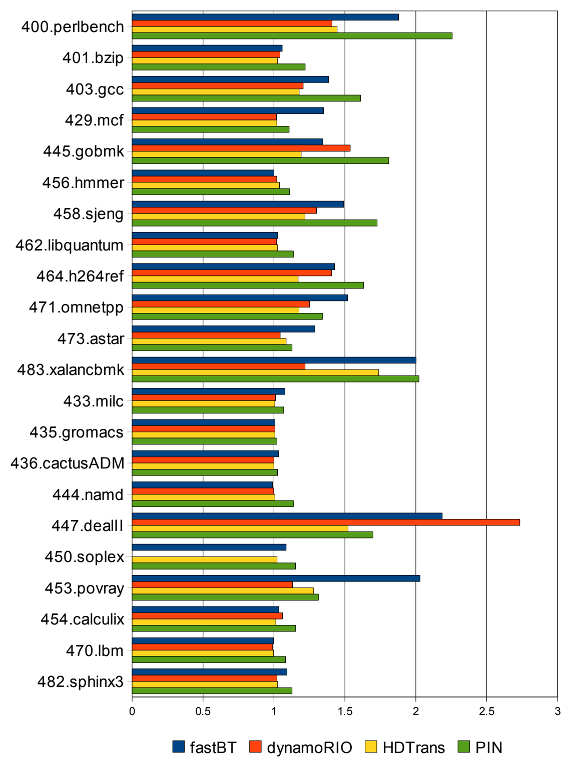

Fig. 8. SPEC CPU2006 benchmarks performance compared to other systems (relative to untranslated code)

All binary translators need some way to handle indirect branches efficiently. Hiser et al. [15] cover different indirect branch mechanisms like lookup inlining, sieves, and a translation cache.

A related topic is full system translation or virtualization. QEMU [3] provides full system *cross machine* virtualization using dynamic translation. VMWare [11; 8] is a full system virtualization tool that uses dynamic translation for executed privileged instructions in supervisor mode, unlike Xen [2] that uses para-virtualization, replacing the privileged instructions already at the source level.

A. HDTrans

HDTrans [22; 21] is a light-weight, table-based instrumentation system. A code cache is used for translated code as well as trace linearization and optimizations for indirect jumps. HDTrans resembles fastBT most closely with respect to speed and implementation, but there are significant differences.

In addition to the support of the sysenter instruction, fastBT raises the level of interaction with the translation system. HDTrans requires a user to supply a low-level translation table that specifies the correct action for each translated instruction; fastBT offers a high-level interface using the table generator. The high-level interface makes it possible to use groups of instructions or properties (e.g., accesses to memory, special-function instructions) as well as individual instructions. HDTrans requires the user to specify any translation directly on the machine-code representation.

Compared to fastBT, HDTrans translates longer chunks of code at a time, only stopping at conditional jumps or return instructions. This strategy can result in longer stalls for the program. Trampolines to start the translator for not already translated targets are inserted into the basic blocks itself. Therefore the memory regions for these instructions cannot be recycled after the target is translated.

B. Dynamo and DynamoRIO

Dynamo is a dynamic optimization system developed by Bala et al. [1]. DynamoRIO [5; 6; 13] is the binary-only IA-32 version of Dynamo for Linux and Windows. The translator extracts and optimizes traces for hot regions. Hot regions are identified by adding profiling information to the instruction stream. These regions are converted into an IR, optimized and recompiled to native code.

Compared to fastBT, DynamoRIO incurs a high foot print (due to profiling, IR extraction, and recompilation) and it remains unclear whether the proposed optimizations offer a significant benefit over the simple table based translation approach because the overhead for profiling, trace selection, as well as IR generation and optimization must be compensated.

C. PIN

PIN[17] is an example of a dynamic instrumentation system that exports a high-level instrumentation API that is available at runtime. The system offers an online high-level interface to all instructions. PIN uses the user-supplied definition and dynamically instruments the running program.

In contrast to PIN, fastBT offers the high-level interface at compile time. A table-based translator that is then used at runtime is generated using the high-level interface. This limits fastBT's alteration possibilities at runtime but removes unneeded flexibility. To keep the design simple fastBT also does not implement transformations like register reallocation, although they could be added to the fastBT framework.

VI. Concluding remarks

We present fastBT, a low overhead, simple binary translator that yields good performance: the translation is fast and the translated program executes efficiently. fastBT is highly configurable at compile time using a table generator, so a user can express the translation at a high level and focus on customizing the translation actions to avoid execution inefficiencies. Only the generated multilevel opcode tables are used at runtime to translate instructions from the user program. This approach to generate a binary translator results in a lean translation engine offering a more flexible interface than a low-level table.

Indirect control transfers limit performance of software-based translations. Although the amount and severity of these control transfers can be reduced through optimizations, they cannot be overcome with software smartness alone.

Nevertheless, such overhead can be tolerated; as demonstrated by fastBT's performance, the execution penalty introduced into the translated program is tolerable. Since fastBT is easily extendable with new optimizations, it provides an attractive platform to investigate approaches to binary translation *and* to generate realistic efficient translator engines. As binary translation is recognized as an important part of the software development tool chain, a table-based generator like fastBT provides the best approach to unify profiling of existing applications, runtime software modifications, as well as debugging and instrumentation of complete programs.

References

[1] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: a transparent dynamic optimization system. In *PLDI '00* (Vancouver, BC, Canada, 2000), pp. 1–12.

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *SOSP '03* (New York, NY, USA, 2003), pp. 164–177.

[3] Bellard, F. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proc. conf. USENIX Ann. Technical Conf.* (Berkeley, CA, USA, 2005), pp. 41–41.

[4] Bruening, D., and Amarasinghe, S. Maintaining consistency and bounding capacity of software code caches. In *CGO '05* (Washington, DC, USA, 2005), pp. 74–85.

[5] Bruening, D., Duesterwald, E., and Amarasinghe, S. Design and implementation of a dynamic optimization framework for windows. In *ACM Workshop Feedback-directed Dyn. Opt. (FDDO-4)* (2001).

[6] Bruening, D., Garnett, T., and Amarasinghe, S. An infrastructure for adaptive dynamic optimization. In *CGO '03* (Washington, DC, USA, 2003), pp. 265–275.

[7] Bruening, D., Kiriansky, V., Garnett, T., and Banerji, S. Thread-shared software code caches. In *CGO '06* (Washington, DC, USA, 2006), pp. 28–38.

[8] Bugnion, E. Dynamic binary translator with a system and method for updating and maintaining coherency of a translation cache. US Patent 6704925, March 2004.

[9] Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, S. B., and Yates, J. Fx!32: A profile-directed binary translator. *IEEE Micro 18*, 2 (1998), 56–64.

[10] Deutsch, L. P., and Schiffman, A. M. Efficient implementation of the smalltalk-80 system. In *POPL '84* (New York, NY, USA, 1984), pp. 297–302.

[11] Devine, S. W., Bugnion, E., and Rosenblum, M. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US Patent 6397242, May 2002.

[12] Garg, M. Sysenter based system call mechanism in linux 2.6 (http://manugarg.googlepages.com/ systemcall-inlinux2_6.html).

[13] Hazelwood, K., and Smith, M. D. Managing bounded code caches in dynamic binary optimization systems. *TACO '06: ACM Trans. Arch. Code Opt. 3*, 3 (2006), 263–294.

[14] Hiser, J., Kumar, N., Zhao, M., Zhou, S., Childers, B. R., Davidson, J. W., and Soffa, M. L. Techniques and tools for dynamic optimization. In *IPDPS* (2006).

[15] Hiser, J. D., Williams, D., Hu, W., Davidson, J. W., Mars, J., and Childers, B. R. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO '07* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 61–73.

[16] Kistler, T., and Franz, M. Continuous program optimization: Design and evaluation. *IEEE Trans. Comput. 50*, 6 (2001), 549–566.

[17] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05* (New York, NY, USA, 2005), pp. 190–200.

[18] May, C. Mimic: a fast system/370 simulator. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques* (New York, NY, USA, 1987), pp. 1–13.

[19] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07* (New York, NY, USA, 2007), pp. 89–100.

[20] Olszewski, M., Cutler, J., and Steffan, J. G. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07* (Washington, DC, USA, 2007), pp. 365–375.

[21] Sridhar, S., Shapiro, J. S., and Bungale, P. P. Hdtrans: a low-overhead dynamic translator. *SIGARCH Comput. Archit. News 35*, 1 (2007), 135–140.

[22] Sridhar, S., Shapiro, J. S., Northup, E., and Bungale, P. P. Hdtrans: an open source, low-level dynamic instrumentation system. In *VEE '06* (New York, NY, USA, 2006), pp. 175–185.

[23] Wen-Ke Chen, Sorin Lerner, R. C., and Gillies, D. M. Mojo: A dynamic optimization system. In *ACM Workshop Feedback-directed Dyn. Opt. (FDDO-3)* (2000).