Generating Low-Overhead Dynamic Binary Translators

Mathias Payer
ETH Zurich, Switzerland
mathias.payer@inf.ethz.ch

Thomas R. Gross
ETH Zurich, Switzerland
trg@inf.ethz.ch

Abstract

Dynamic (on the fly) binary translation is an important part of many software systems. In this paper we discuss how to combine efficient translation with the generation of efficient code, while providing a high-level table-driven user interface that simplifies the generation of the binary translator (BT).

The translation actions of the BT are specified in high-level abstractions that are compiled into translation tables; these tables control the runtime program translation. This table generator allows a compact description of changes in the translated code.

We use *fastBT*, a table-based dynamic binary translator that uses a code cache and various optimizations for indirect control transfers to illustrate the design tradeoffs in binary translators. We present an analysis of the most challenging sources of overhead and describe optimizations to further reduce these penalties. Keys to the good performance are a configurable inlining mechanism and adaptive self-modifying optimizations for indirect control transfers.

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

General Terms Languages, Performance, Optimization, Security

Keywords Dynamic instrumentation, Dynamic translation, Binary translation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SYSTOR 2010 May 24–26, Haifa, Israel.
Copyright © 2010 ACM X-XXXXX-XXX-X/YY/MM. . . $5.00

1. Introduction

Binary translation (BT) is the art of adding, removing or replacing individual instructions of the translated program. Two different approaches to BT exist: (i) static, ahead-of-time translation and (ii) dynamic, just-in-time translation at runtime. BT enables program instrumentation and makes it possible to, e.g., add dynamic safety and validity checks to profile and validate a binary program, extend the binary program to include new functionality, or to patch the program at runtime. Another aspect of BT includes virtualization, where code is encapsulated in an additional layer of protection and all interactions between the binary code and the higher levels are validated.

The key advantage of static translation is that the translation process does not incur a runtime penalty and can be arbitrarily complex. The principal disadvantage is that static BT is limited to code regions that can be identified statically. This approach is problematic for dynamically loaded libraries, self-modifying code, or indirect control transfers. Dynamic BT translates code right before it is executed the first time. Dynamic BT adapts to phase changes in the program and optimizes hot regions incrementally. This paper focuses 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. An important design decision is that the stack of the user program remains unchanged. This principle makes virtualization possible where the program does not recognize that it is translated. The untranslated return addresses result in an additional lookup overhead but are needed for some programs (e.g., for exception management, debugging, and self-modifying code).

The two key aspects in designing a fast binary translator are a lean translation process and low additional runtime overhead for any translated code region. Three forms of indirect control transfers are the source of most of the dynamic overhead:

1. **Indirect jumps:** The target of the jump depends on dynamic information (e.g., a memory address, or a register) and is not known at translation time. Therefore a runtime lookup is needed.
2. **Indirect calls:** Similar to indirect jumps the target is not known and a runtime lookup is needed for every execution.
3. **Function returns:** The target of this indirect control transfer is on the stack and the stack contains only untranslated addresses, therefore a runtime lookup is needed.

This paper presents fastBT, a generator for low-overhead, low footprint, table-based dynamic binary translators with efficient optimizations for all forms of dynamic control transfers (e.g., novel combined predictor and fast dynamic control transfer schemes that adaptively select the optimal configuration for each indirect control transfer, or the fastRET return optimization). We argue that a single optimization will not suffice for a class of control transfers, but depending on the arguments and the encoding the best fitting optimization depends on the individual code location. Depending on the runtime behavior of the application a less than optimal fit is adaptively replaced by a different strategy at runtime. This results in an optimal translation on a per location basis for indirect control transfers and not on a per class strategy. Additionally fastBT offers a novel high-level interface to the translation engine based on the table generator without sacrificing performance.

fastBT's design and implementation are architecture-neutral but the focus of this paper is on IA-32 Linux. The current implementation provides tables for the Intel IA-32 architecture, and uses a per-thread code 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 other translators. 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, or allows to redirect all system calls to an authorization framework that encapsulates and virtualizes the user program.

In this paper we present the general infrastructure and methods to optimize indirect control transfers, thereby reducing the overhead of dynamic BT. We show different optimizations for function returns and indirect jumps/calls. These evaluations are made using a low-overhead binary translator: When comparing programs transformed by fastBT with the original program using a "null" transformation (e.g., a transformation without additional instrumentation, checks, or statistics compared to the original program) the transformed programs exhibit a small speedup or a minor slow-down (between -3% to 5% for the majority of benchmarks, i.e. 17 programs), noticeable overhead of 6% to 25% for 9 benchmarks, and two outliers with an overhead of 44% and 56%. The average overhead introduced through dynamic BT is 6% for the SPEC CPU2006 benchmarks. As part of the investigation of different optimization strategies, we also compare fastBT with HDTrans, PIN, and DynamoRIO; fastBT always outperforms PIN and HDTrans in most benchmarks. The comparison with DynamoRIO depends on application characteristics and is discussed later in depth.

2. A generic dynamic binary translator

A generic dynamic binary translator processes basic blocks of input instructions, places the translated instructions into a code cache, and adds entries to the mapping table. The mapping table is used to map between addresses in the original program and the code cache. Only translated code of the original program that is placed in the code cache is executed. The translator is started and intercepts user code whenever the user program branches to untranslated code. The user program is resumed as soon as the target basic block is translated. See Figure 1 for an overview of such a basic translator.

The user stack contains only untranslated instruction pointers. This setup is needed to support exception management, debugging, and self-modifying code. Each one of these techniques accesses the program

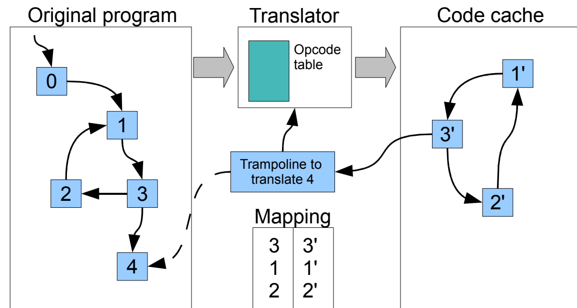

Figure 1: Runtime layout of the binary translator. Basic blocks of the original program are translated and placed in the code cache using the opcode tables.

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 code cache, instead of into the user program. Therefore fastBT replaces every return instruction with an indirect control transfer to return to translated code. This design decision follows our user-space virtualization principle to not let the program know that it is translated.

2.1 Translation tables

The translation engine is a simple table-based iterator. The translator is invoked with a pointer to an untranslated basic block, allocates space for the translated instructions, adds an entry into the mapping table, and translates the basic block one instruction at a time using the translation tables.

Instructions of the IA-32 architecture have a variable length from 1 to 16 bytes. These instructions are decoded using multidimensional translation tables. The translation tables contain extensive information about all possible encodings for all IA-32 instructions and parameters. The decoding of a new instruction starts with the first byte of the instruction that is used as an index into the first translation table. The indexed row contains information about the instruction or a forwarding pointer to the next table with the next instruction byte if the instruction is not finished. The decoding process continues until the instruction is decoded. The final indexed row contains the information about the instruction. Table 1 shows an example.

After the decoding of the instruction the instruction and parameters are passed to the corresponding action function. The action function handles the transcription

...	...	...
`flags`	`*next_tbl (or NULL)`	`&action_func`
...	...	...

Table 1: Layout of an opcode table, the opcode byte is used as table offset. The flags contain information about parameters, immediate bytes, and registers.

of the instruction to the code cache. The function can alter, copy, replace, or remove the instruction.

The translation process stops at recognizable basic block boundaries like conditional branches, indirect branches, or return instructions. Backward branches are not recognizable by a single-pass translator, in this case the second part of the basic block is translated a second time. The additional translation overhead for these split basic blocks is negligible and smaller than the overhead of a multi-pass translator.

At the end of a basic block the translator checks if the outgoing edges are already translated, and adds jumps to the translated target. For untranslated targets a trampoline is constructed that starts the translation of the target. Trampolines are short sequences of code that transfer control flow to a different location.

2.2 Predefined actions

A translator needs different action functions to support the identity transformation. The main challenge is to redirect control flow into the code cache but contain the illusion as if the program was executed from the original location.

Simple action functions remove the instruction or copy the instruction into the code cache. Other actions are needed to keep the execution flow inside the code cache:

action_jmp: Translates a direct jump in the original program. This action uses the mapping table to look up the translated target and encodes a branch to the translated target into the code cache.

action_call: This action emits code to push the original location onto the stack and emits a branch to the translated call target.

action_jcc: Translates a conditional jump. Code that branches to the translated jump targets is emitted into the code cache. A trampoline is constructed for untranslated targets.

action_jmp_ind: This action encodes a runtime lookup that translates the original target into a target in

the code cache and emits a branch to the translated target.

action_call_ind: Emits code that pushes the original location onto the stack followed by a runtime lookup and dispatch of the indirect target into a location in the code cache.

action_ret: This action emits a translation of the return address on the stack into a target in the code cache.

2.3 Code cache

It is likely that code regions are executed multiple times during the runtime of a program. Therefore it makes sense to keep translated code in a code cache for later reuse. Code is translated only once, reducing the overall translation 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 thread-local traces.

Important advantages of thread local caches are that (i) accesses to the code cache need not be synchronized between threads and the high overhead of locking is avoided, and (ii) it is possible to emit hard-coded pointers to thread local data structures which remove the need to call lookup functions. 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.

A trampoline is a piece of code that triggers the translation of a basic block. fastBT places these trampolines in a separate code region where they can be reused after the target's translation. This keeps the code cache clean and increases code locality.

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

2.4 Mapping table

The mapping table is a hash map that maps addresses in the original program to addresses in the code cache. The hash map contains all translated basic blocks.

The original program does not know that it is translated, so all targets for indirect control transfers (e.g., function returns, indirect jumps, and indirect calls) must be handled through an online lookup and dispatch. These dispatch mechanisms use the mapping table to get the corresponding translated target in the code cache. The translator uses the mapping table whenever it translates function calls, conditional branches, and direct branches.

The hash function that is used in these lookup functions returns a relative offset into the mapping table. This hash function is kept as simple as possible because (i) perfect or near perfect hashing is not needed, (ii) the overhead for hashing should be low, and (iii) it should be implemented in a few machine code instructions. In the case of a collision the hash function loops through the mapping table until the entry or NULL is found. Tests with SPEC CPU2006 benchmarks and various programs showed that the simple hash function `addr<<3 & (HASHTABLE_SIZE-1)` results in a low number of collisions. Note that the hashtable size must be 2^x bytes. A good value for x is 23, which results in an 8MB hashtable that holds up to 2^{20} individual address mappings.

2.5 Signal and syscall handling

User-space binary translation systems 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 so that the binary translator regains control after the context switch.

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

3. Table generator

The generic translator uses a multilevel translation table with information about each possible machine code instruction. Writing these translation tables by hand is hard and error prone as many different combinations of instructions and prefixes exist.

fastBT uses a two-level process to offer the programmer a high-level interface. A table generator uses high-level opcode tables and generates the low-level trans-

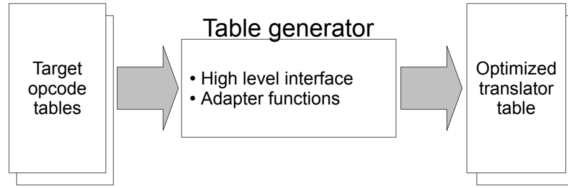

Figure 2: Compile time construction of the translator table.

lation tables. The low-level translation tables are then used at runtime.

Figure 2 shows a schema of the compile-time table generation process. 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. Additional action functions that are referenced at compile-time can then be added to the runtime translator where they are used in the translation process.

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.

```
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);
```

Figure 3: Example for an adapter function that redirects all instructions that access memory to a special action - C++ Code.

Using adapter functions, the user can specify which types or instructions are instrumented 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 and 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 that is used in a simple software transactional memory system. The output of the table generator is then compiled and linked together with the fastBT sources to the binary translator library. This library is then injected into the user program.

4. fastBT optimizations

Indirect control transfers are the highest source of overhead in a dynamic binary translator. Code translation and memory management are amortized by multiple executions of the code. Indirect control transfers incur overhead every time they are executed. The optimizations with the biggest effect in fastBT are those targeting indirect control transfers.

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

Figure 4: Naive translation of indirect jumps/calls and return instructions. The (translated) target of indirect control transfers is not known at translation time, so the target must be looked up at runtime. *tld* is thread local data.

Figure 4 shows a simple translation of the different indirect control transfers. The naive approach uses the `ind_jump` routine to translate control transfers which is responsible for a large portion of the runtime overhead. The indirect jump function maps and dispatches an address in the user program to an address in the code cache. This section discusses the different optimizations for indirect control transfers used in fastBT. These optimizations replace the call to `ind_jump` and reduce the overhead through predictions, inlining, and fast lookups.

4.1 Indirect jump/call optimizations

Indirect jumps and indirect calls are very similar. Both result in a runtime lookup in the mapping table and a control transfer to a dynamic location. Additionally the indirect call pushes its source location onto the stack.

fastBT uses three different optimizations for indirect control transfers: (i) a fast indirect control transfer that is inlined into the code cache, (ii) a prediction that compares the target with the last target and caches the last destination, and (iii) a combination of both.

4.1.1 Fast indirect control transfer

The fast indirect control transfer is a set of instructions that is inlined into the code cache. These instructions execute a lookup into the mapping table and dispatch to the corresponding basic block in the code cache. Figure 5 shows an inlined control transfer for indirect jumps. If an indirect call is translated then the `pushfl` instructions can be omitted and a `pushl $srcip` must be prepended.

```
 pushfl
 pushl %ebx, %ecx
 movl 12(%esp), %ebx # load target
 movl %ebx, %ecx
 andl HASH_PATTERN, %ebx
 cmpl maptbl_start(0,%ebx,8), %ecx
 jne nohit
 movl maptbl_start+4(0,%ebx,8), %ebx
 movl %ebx, tld->jumptarget
 popl %ecx, %ebx
 popfl
 jmpl *(tld->jumptarget)
nohit:
 popl %ecx, %ebx
 popfl
 pushl $tld
 call ind_jmp
```

Figure 5: A translated fast indirect control transfer. If the mapping table lookup is successful, execution is redirected to the translated counterpart, otherwise an `ind_jump` is used to recover. *tld* is the pointer to thread local data.

Using this optimization an indirect jump in the source program is translated to 14 instructions and an indirect call in the source program is translated to 13 instructions if the mapping table lookup is successful. Otherwise control is transfered from the code cache to the translator, which checks if there was a mapping table miss or if the target needs to be translated first.

4.1.2 Indirect branch prediction

The indirect branch prediction is an optimistic optimization that speculates that the number of different targets for a certain branch location is low. This optimization caches one or two source addresses and destinations locally in the code cache. If the prediction is correct then execution can continue directly at the translated target. Otherwise a fixup routine is executed to update the local cache. Figure 6 shows the different indirect branch predictions for indirect calls and indirect jumps.

Indirect call	Indirect jump
`pushl $srcip`	`pushfl`
`cmpl $ABCD, xx(%src)`	`cmpl $ABCD, xx(%src)`
`je $transAABBCCDD`	`jne fixup`
`cmpl $EF01, xx(%src)`	`popfl`
`je $transEF01`	`jmp $transABCD`
`pushl xx(%src)`	`fixup: popfl`
`pushl $tld`	`pushl xx(%src)`
`pushl $addrOf(ABCD)`	`pushl $tld`
`pushl $addrOf(EF01)`	`pushl $addrOf(ABCD)`
`call fix_ind_call_pred`	`call fix_ind_jmp_pred`

Figure 6: A translated indirect control transfer using the prediction optimization. The prediction compares the indirect target with the local cache. If there is no cache hit then a fixup routine is used to recover. *tld* is the pointer to thread local data.

If the prediction is correct then an indirect jump is translated to 5 instructions and an indirect call to 3 or 5 instructions depending on whether the first or second prediction is correct. Otherwise a fixup routine is executed which updates the prediction with the current source address and destination. For the indirect call prediction the least recently updated entry gets discarded.

4.1.3 Adaptive combined optimization

The indirect branch prediction is an optimistic optimization. Depending on the usage of indirect control transfers in the program and the number of branch targets the prediction can outperform a fast indirect control transfer or fall behind if the miss-rate is high.

The combined optimization for indirect control transfers adds a counter for the number of mispredictions per indirect control transfer. After a certain amount of mispredictions the prediction is removed and replaced by a fast indirect control transfer.

This optimization offers the best performance on average. All programs that we analyzed have some locations where the prediction is faster than the fast indirect control transfer, but there are also locations which suffer under a high miss-rate. The combined optimization optimizes indirect control transfers on a per location basis and not on a global, compile time selection of a single optimization. The adaptive combined optimiza-

tion adaptively selects the best optimization *for each location*.

4.1.4 Shadow jump table

The shadow jump table optimization is a novel optimization for indirect jumps. This optimization is applied only to a subset of indirect jumps that look as if they used a jump table (e.g., `jmpl *addr(, %reg, 4)`, whereas `reg` is any register and `addr` is the base address of a jump table).

For such jumps a shadow jump table is constructed that contains translated locations instead of locations in the user program. The indirect jump is then prepended by a bound check, and the base address of the indirect jump is replaced by the new shadow jump table. The size of the jump table cannot be determined at compile time, so the translated jump must check the bounds. Figure 7 shows a translated indirect jump redirected through a shadow jump table.

```
 pushfl
 cmpl JUMPTABLE_SIZE, %reg
 jge jmpind
 popfl
 jmpl shadow_base(,%reg, 4)
jmpind:
 popfl
 pushl jmp_base(, %reg, 4)
 pushl $tld
 call ind_jmp
```

Figure 7: A translated indirect jump which uses the shadow jump table optimization. The shadow jump table is constructed at translation time and values are patched in whenever a location is used the first time. *tld* is the pointer to thread local data.

This optimization translates an indirect jump into 5 instructions if the entry in the shadow jump table is already translated and the jump table is smaller than the maximum size. This optimization covers most of the cases where the prediction optimization does not suffice and is able to outperform the fast indirect control transfer.

4.2 Function call optimizations

Return instructions are responsible for a major fraction of the instrumentation overhead. fastBT offers several different optimization strategies for return instructions. These optimizations reduce the costs of the indirect control transfers.

The call/ret relationship offers an optimization opportunity, because the return address is already known at the time of the call. The return cache optimization tries to exhibit this duality and decreases the number of executed instructions in the best case.

The indirect branch prediction does not work for return instructions although they are similar to indirect jumps. The return instruction transfers control flow to the address that is on the top of the stack. Many functions are called from multiple locations, therefore a simple predictor does not work. Even a combined predictor with fastRET is still slower than fastRET alone.

4.2.1 fastRET

The fastRET optimization applies the idea of a fast indirect control transfer to return instructions (see Section 4.1.1). Similar code as for an indirect call is emitted into the code cache that executes a lookup and dispatch of the address that is on the stack. This translates a return instruction into 12 instructions in the code cache.

4.2.2 Return cache

The return cache optimization uses the duality between a function call and the following return instruction and is somewhat similar to HDTrans' return caching.

The call instruction pushes the return instruction onto the stack, updates the thread local return cache and performs a jump to the translated call target. The translation of the return instruction performs an indirect jump through the return cache. These return cache targets check if the top of the stack matches their prediction. If the prediction does not match a fast indirect jump is used to recover, otherwise execution continues at the correct location.

Return instruction	Return trampoline
`pushl %ebx`	`cmpl 4(%esp), rip`
`movzbl 4(%esp), %ebx`	`jz hit`
`jmpl *rc(, %ebx, 4)`	`popl %ebx`
	`call ind_jmp`
Call instruction	`hit: popl %ebx`
`movl $ret_tramp, rc+offs`	`leal 4(%esp), %esp`

Figure 8: Translated return and call instructions and the corresponding trampoline that is referenced through the return cache, *rc* is the address of the return cache, *rip* is the return instruction pointer.

Figure 8 shows a translated return instruction and the corresponding trampoline that is reached through the

earlier indirect jump. This optimization results in 7 instructions if the cache contains the correct trampoline. If the `call` and `ret` instructions do not match (e.g., due to recursion) an additional indirect jump is executed.

Differences to HDTrans are that (i) fastBT uses a generic return trampoline for the call instrumentation that is backpatched only as soon as the return location is translated, and (ii) the protocol expects that the `ebx` register is already pushed and can be used in the return trampoline and during the fixup phase, which saves some instructions if there is no cache hit.

4.2.3 Function inlining

Function inlining is an effective way of reducing the overhead of indirect control transfers. If a function is inlined then the return instruction can be translated as an addition of 4 to the `esp` register instead of an indirect jump. This saves at least 12 instructions compared to the fastRET optimization.

5. Performance evaluation

This section shows an in depth performance evaluation of fastBT, including an analysis of individual configurations and collected runtime statistics.

The benchmarks were run on an Ubuntu 9.04 system with an E6850 Intel Core2Duo CPU running at 3.00GHz, 2GB RAM, and GCC version 4.3.3.

The SPEC CPU2006 benchmark suite version 1.0.1 is used to evaluate the single threaded performance. We compare fastBT with three other binary translators: HDTrans [24] version 0.4.1, PIN [18] revision 29972, and DynamoRIO [5] version 1.4.0-20.

Averages in the tables are calculated by comparing overall execution time for all programs of untranslated runs against translated runs.

5.1 Instrumentation overhead

Figure 9 shows different configurations of fastBT and compares the resulting overhead. The configuration names are composed of optimizations for return instructions (Rx), indirect call instructions (Cx), and indirect jump instructions (Jx). Surveyed optimizations for x are F for fast control transfers, P for predicting optimizations, and C and Mult for combined optimizations. The different configurations are:

1. **RFCFJF:** A fast indirect control transfer is used for return instructions, indirect calls, and indirect jumps.

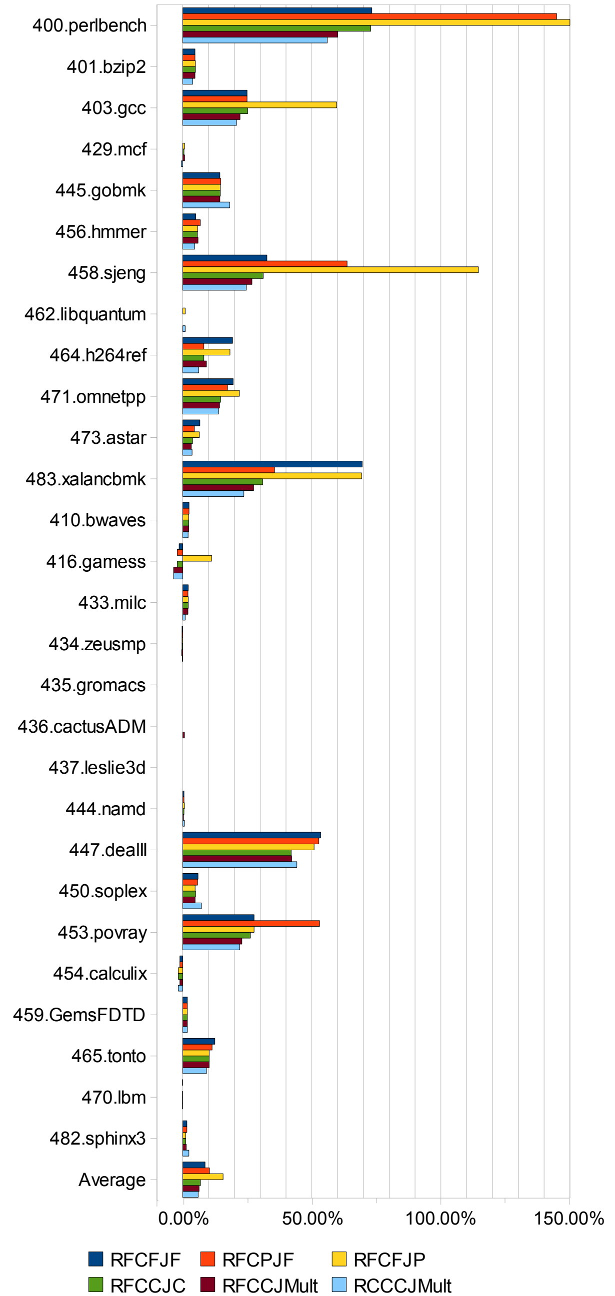

Figure 9: Overhead of different fastBT configurations relative to an untranslated run. Lower is better. The graph for 400.perlbench has been cut off for RFCFJP, the overhead for this outlier is 227.16%.

2. **RFCPJF:** A fast indirect control transfer is used for return instructions and indirect jumps. A predictor is used for indirect calls.

3. **RFCFJP:** A fast indirect control transfer is used for return instructions and indirect calls. A predictor is used for indirect jumps.

4. **RFCCJC:** The fastRET optimization is used together with the combined indirect call and jump optimization.
5. **RFCCJMult:** The fastRET optimization is used together with the combined indirect control transfer optimization for indirect calls and indirect jumps and the jump table optimization for suited indirect jumps.
6. **RCCCJMult:** This configuration combines the combined optimization for indirect calls and indirect jumps, the jump table optimization for suited indirect jumps, and the return cache for return instructions.

The indirect call prediction 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.

The comparison of the different optimization strategies shows that different benchmarks profit from different optimizations. Benchmarks written in object oriented languages profit from the indirect call prediction as this optimization enables fast virtual calls without a mapping table lookup. Other benchmarks like 400.perlbench profit from the jump table optimization that speeds up switch statements that are used often in interpreters.

The RFCFJP configuration shows bad performance due to some indirect jump locations that incur a high number of mispredictions. But the combined optimization selects the best optimization for each individual location so that the predictor can reduce overhead for indirect jumps as well. The statistics show that the number of removed predictions is low for all benchmarks.

The most important result of the comparison in Figure 9 is that no single optimization can work for a class of indirect control transfers, but the binary translator must adapt and select the best possible optimization for each individual translated indirect control transfer. The combined optimizations of fastBT are able to adaptively select the most promising configuration on a per location basis and perform well in practice.

5.2 Difference to fastBT 0.1

An earlier version of fastBT [22] used a combination of function inlining, an earlier version of fastRET, fast indirect control transfers for indirect jumps, and a single prediction for indirect calls.

Benchmark	fastBT 0.1	fastBT
400.perlbench	88%	56%
403.gcc	39%	21%
445.gobmk	34%	18%
458.sjeng	49%	25%
464.h264ref	43%	6.2%
471.omnetpp	52%	14%
483.xalan	100%	24%
447.dealII	99%	44%
453.povray	103%	22%

Table 2: Comparison of the old fastBT version to the current optimizations for challenging benchmarks and an average overhead for the benchmarks with overhead above 5%.

This combination offered good average performance and low overhead, but the combined predictions and fast control transfers, the jump table optimization for indirect jumps and the return cache lower the average overhead further. Table 2 offers an overview of the most challenging benchmarks with the highest overhead.

The additional optimizations and combined optimizations in the new version of fastBT show that it is important to separate between predictable indirect control transfers with only a few target locations and other transfers which incur a high miss-rate. The online analysis makes it possible to select the optimal optimization for each individual indirect control transfer.

5.3 Benchmark statistics and classification

Table 3 shows detailed information of the different SPEC CPU2006 benchmarks. For some benchmarks inlining removes up to 85.1% of the function calls and reduces the overhead significantly. For the remaining benchmarks the return cache reduces the overhead in most cases (between 0% and 48.9% of the calls are redirected to the fast return optimization).

The table shows that for indirect jumps the jump table optimization and the prediction cover most of the indirect control transfers, and the fast redirection is seldom needed.

For the majority of benchmarks the indirect call prediction is sufficient. But some benchmarks show poor performance for indirect calls. Further optimizations are needed to reduce the overhead of indirect calls for benchmarks like 400.perlbench, 458.sjeng, and 453.povray.

Benchmark	Function calls	(inlined)	(retc miss)	Indirect jumps	(jmptbl)	(pred)	Indirect calls	(pred)
400.perlbench	25814694843	8.1%	22.7%	21930067960	93.7%	6.3%	3903053407	7.4%
401.bzip2	6680608156	0.0%	28.2%	1869734	8.8%	91.2%	2076	100.0%
403.gcc	11731029096	2.7%	20.4%	5275640406	84.8%	10.1%	653683244	52.9%
429.mcf	6945269263	0.0%	0.1%	1708543	0.0%	100.0%	574680	100.0%
445.gobmk	18001407703	1.3%	33.9%	93605731	1.0%	99.0%	185811548	4.1%
456.hmmer	212603875	27.6%	3.0%	163062236	39.0%	61.0%	1138925	100.0%
458.sjeng	24769309279	1.1%	13.6%	10992929516	97.5%	2.5%	5070023361	0.0%
462.libquantum	1762604025	50.0%	0.0%	784	0.0%	100.0%	249	1.6%
464.h264ref	37463457335	7.4%	4.0%	1189567749	1.1%	98.9%	28445058263	99.2%
471.omnetpp	19737972588	18.0%	33.1%	3153780227	21.3%	78.6%	2733908888	89.8%
473.astar	15647809660	35.2%	32.0%	10808740	0.0%	100.0%	4996558207	100.0%
483.xalancbmk	28888952021	10.6%	17.7%	2627706146	27.0%	63.6%	9161680928	96.1%
410.bwaves	1270922878	85.1%	0.0%	216329663	0.0%	100.0%	4537	90.9%
416.gamess	2778622627	35.2%	33.0%	1889868887	74.1%	25.8%	820281	100.0%
433.milc	6682737588	1.4%	11.2%	11999701	32.0%	68.0%	3856876	100.0%
434.zeusmp	96755	38.2%	10.3%	20806	0.0%	82.2%	3084	100.0%
435.gromacs	3339398623	78.9%	4.8%	27898268	2.6%	95.7%	3274864	99.1%
436.cactusADM	1668709640	0.5%	0.4%	1650774821	0.0%	100.0%	223605	100.0%
437.leslie3d	6419497	50.7%	18.2%	2063899	0.0%	98.5%	88425	100.0%
444.namd	27828322	24.6%	5.0%	16777617	0.0%	88.9%	1969171	100.0%
447.dealII	52756915715	54.5%	38.8%	21147326182	1.7%	98.3%	540282822	98.4%
450.soplex	2537751419	3.5%	3.3%	1707604759	83.8%	16.2%	27482402	100.0%
453.povray	25743213080	5.9%	28.6%	456989748	34.9%	65.0%	7072491440	5.0%
454.calculix	5193872239	25.5%	26.6%	502409545	2.1%	95.7%	11962078	100.0%
459.GemsFDTD	4527866169	50.3%	48.9%	1510763422	0.0%	99.9%	11839192	100.0%
465.tonto	28170484438	50.2%	21.5%	9572766779	7.6%	92.4%	2648364	100.0%
470.lbm	5268354	50.0%	0.0%	2627702	0.0%	100.0%	3761	100.0%
482.sphinx3	7456046206	10.6%	21.2%	273089361	0.0%	98.4%	6126896	100.0%

Table 3: Detailed per benchmark statistic. Out of the absolute number of function calls *(inlined)* are executed inline and *(retc miss)* function calls miss the return cache. The absolute number of indirect jumps per benchmark is divided into the percentage that is executed through a jump table *(jmptbl)* and through a prediction *(pred)*. The remaining indirect jumps are executed through a fast indirect jump. The overall number of indirect calls is executed through predictions *(pred)*, the remaining calls are executed with the fast indirect call.

The detailed statistics show that for these three benchmarks only a small number of indirect calls is responsible for a high percentage of overall calls. For 400.perlbench 24 out of 322 translated calls fall back to the fast indirect call but are responsible for 92.6% of the calls. For 458.sjeng one out of 43 translated indirect calls is responsible for almost 100% of the fast indirect calls. For 453.povray eight out of 112 indirect calls fall back to the fast indirect call and are responsible for 95% of the overall indirect calls.

These information show that only a small fraction of indirect calls is responsible for most of the remaining indirect call overhead. A possible future optimization could target the low predictor rate for some indirect calls.

5.4 Comparative performance

Table 4 compares the overhead of fastBT to PIN, HDTrans, and DynamoRIO. fastBT outperforms PIN in every benchmark and HDTrans in most benchmarks. The 400.perlbench benchmark is the only benchmark where the overhead of HDTrans is significantly lower than the fastBT overhead (44% for HDTrans versus 56% for fastBT). Part of the additional overhead results from the missed optimization opportunity for indirect calls that exhibit a low correct prediction rate for the 400.perlbench benchmark.

The 483.xalancbmk benchmark shows that fastBT predicts most of the indirect calls and indirect jumps for benchmarks written in object oriented languages. The resulting overhead for fastBT is 24% compared to 74% for HDTrans, 97% for PIN, and 18% for DynamoRIO.

Benchmark	no BT	fastBT	HDT	PIN	dRIO
400.perlbench	486s	56%	44%	126%	36%
401.bzip2	668s	4%	3%	36%	4%
403.gcc	441s	21%	20%	140%	25%
429.mcf	405s	0%	0%	6%	0%
445.gobmk	611s	18%	19%	91%	32%
456.hmmer	926s	5%	4%	45%	1%
458.sjeng	727s	25%	21%	64%	23%
462.libquantum	1020s	1%	2%	25%	2%
464.h264ref	989s	6%	17%	227%	20%
471.omnetpp	496s	14%	18%	28%	8%
473.astar	601s	4%	7%	21%	2%
483.xalancbmk	371s	24%	74%	97%	18%
410.bwaves	895s	2%	1%	11%	4%
416.gamess	1430s	-3%	-6%	13%	-7%
433.milc	827s	1%	1%	7%	2%
434.zeusmp	793s	0%	0%	6%	0%
435.gromacs	1440s	0%	0%	3%	0%
436.cactusADM	1520s	0%	0%	1%	0%
437.leslie3d	1160s	0%	0%	19%	0%
444.namd	616s	1%	0%	7%	-1%
447.dealII	552s	44%	55%	54%	14%
450.soplex	538s	7%	3%	17%	2%
453.povray	362s	22%	25%	39%	17%
454.calculix	1790s	-2%	-1%	13%	-1%
459.GemsFDTD	1120s	2%	1%	4%	1%
465.tonto	925s	9%	9%	34%	8%
470.lbm	930s	0%	0%	0%	-3%
482.sphinx3	846s	2%	3%	19%	-1%
Average	839s	6%	7%	34%	5%

Table 4: Comparison of overhead introduced through different binary translation systems (fastBT, HDTrans, PIN, and DynamoRIO) using the SPEC CPU2006 benchmarks. No BT denotes the runtime in seconds without BT.

Benchmark	no BT	fBT	HDT	PIN	dRIO
400.perlbench	4s	29%	26%	1415%	308%
401.bzip2	8s	2%	0%	41%	3%
403.gcc	1s	41%	48%	776%	247%
429.mcf	3s	1%	0%	28%	5%
445.gobmk	21s	18%	19%	139%	43%
456.hmmer	6s	6%	5%	46%	0%
458.sjeng	5s	24%	20%	82%	28%
462.libquantum	<1s	33%	11%	453%	100%
464.h264ref	23s	6%	23%	83%	25%
471.omnetpp	1s	66%	142%	363%	74%
473.astar	11s	4%	6%	26%	5%
483.xalancbmk	<1s	56%	139%	3481%	745%
410.bwaves	12s	2%	1%	16%	4%
416.gamess	1s	9%	15%	392%	84%
433.milc	10s	2%	4%	38%	9%
434.zeusmp	21s	0%	0%	14%	1%
435.gromacs	3s	2%	1%	42%	8%
436.cactusADM	4s	0%	0%	37%	7%
437.leslie3d	29s	0%	1%	21%	1%
444.namd	17s	1%	1%	13%	0%
447.dealII	25s	36%	47%	69%	16%
450.soplex	<1s	102%	45%	3836%	929%
453.povray	1s	23%	28%	221%	62%
454.calculix	<1s	59%	45%	2346%	513%
459.GemsFDTD	4s	7%	13%	70%	15%
465.tonto	1s	31%	27%	300%	74%
470.lbm	6s	1%	1%	8%	0%
482.sphinx3	2s	12%	9%	70%	18%
Average	8s	10%	13%	87%	21%

Table 5: Comparison of overhead for short programs introduced through different binary translation systems (fastBT, HDTrans, PIN, and DynamoRIO) using the SPEC CPU2006 benchmarks with the test dataset. No BT denotes the runtime in seconds without binary translation.

The same predictors work for the 464.h264ref benchmark where fastBT has a low overhead of 6% compared to 17% for HDTrans, 227% for PIN, and 20% for DynamoRIO.

The average overhead for fastBT is 6% compared to 7% for HDTrans, 34% for PIN, and 5% for DynamoRIO. These numbers show that fastBT succeeds to combine a high-level interface at compile time with a simple table based translator implementation and powerful optimization to achieve performance similar to a full blown binary optimizer based on intermediate representations like DynamoRIO.

5.5 Translation efficiency

The numbers in 5.4 show translation efficiency of the presented binary translations and overhead introduced through the binary translation process.

Table 5 shows a comparison of overhead for short programs. This comparison uses the test dataset of the SPEC CPU2006 benchmarks to measure the startup and translation overhead of the different binary translators.

It is important that a binary translator produces efficient code, but the overhead of the translation process should be low as well. Only if both properties are fulfilled it is possible to translate programs efficiently.

The average overhead for small programs as shown in Table 5 is 10% for fastBT compared to 13% for HDTrans, 87% for PIN, and 21% for DynamoRIO. These numbers show that fastBT scales well for small programs and that the fastBT translation process is efficient.

It remains unclear whether approaches working with an intermediate representation (IR) offer a significant benefit over the simple table based translation approach because the overhead for profiling, trace se-

lection, as well as IR generation and optimization must be compensated. The numbers show that fastBT competes with IR based solutions for long running benchmarks. Especially for short running programs IR based translators cannot achieve competitive execution times and fastBT outperforms the IR based solutions DynamoRIO and PIN clearly.

6. Related work

Binary translation has a long history [10, 13, 19]. In this section we focus on current binary translators that are similar to fastBT either in the offered interface and the target application or the design and implementation. Dynamic binary translators can be separated into two groups: The first group uses a rewriting scheme based on low-level translation tables (e.g., fastBT, HDTrans, Mojo [25], and JudoDBR [21]). The second group parses the machine code stream into an intermediate representation (IR) and uses common compiler techniques to work on that IR (e.g., DynamoRIO [6], PIN [18], and Valgrind [20]).

IR-based approaches offer the advantage of more sophisticated optimization possibilities, but the IR translation comes at a higher runtime cost, which must be amortized. IR-based translators can use profiles to generate runtime information but there is no structure or type information at the machine-code level. Therefore it is hard to achieve good performance with IR-based translation systems. Some binary translation systems (e.g., [15, 17]) use a dual approach of profiling and adaptive optimization to limit the dynamic compilation overhead. FX!32 [9] uses emulation for infrequently executed code and profile generation and offline static recompilation of hot code to speed up overall execution.

The most important overhead of dynamic binary translators is the handling of indirect control transfers. Hiser et al. [16] cover different indirect branch mechanisms like lookup inlining, sieves, and a translation cache.

A related topic that uses binary translation is full system virtualization. QEMU [3] provides full system cross machine virtualization. VMware [8, 11] is a full system virtualization system that uses dynamic translation for privileged instructions in supervisor mode, unlike Xen [2] that uses para-virtualization, which replaces privileged instructions at the source level.

6.1 HDTrans

HDTrans [23, 24] 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.

HDTrans requires hand-coded low-level translation tables that specify translation properties and actions for every single instruction. This setup requires a very deep understanding of the translator and relations between different optimizations. fastBT raises the level of interaction with the translation system. Low-level translation tables are generated using a table generator that specifies transformations on a high level. Instructions transformations can be specified on their properties (e.g., memory access, specific registers, special-function instructions, or extension groups like SSE and FPU), or on an instruction opcode.

To make the indirect jumps faster HDTrans uses a hash table of jump code blocks called sieve. The target is hashed and then a jump into the sieve is issued. The code block in the sieve then takes care of the jump to the correct location whereas fastBT uses a simpler hash table and hand optimized assembly code that is emitted into the code cache. Instead of a sieve fastBT uses a lookup table that maps locations in the code cache to locations in the original program.

Return instructions are handled differently from normal indirect control transfers to take care of the `call/ret` relationship. HDTrans uses a (small) direct mapped cache to speed up return instructions, with a fall-back to the sieve. fastBT uses a similar approach with the return cache that uses a bidirectional protocol.

Compared to fastBT, HDTrans translates longer chunks of code at a time, stopping only 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.

6.2 Dynamo and DynamoRIO

Dynamo is a dynamic optimization system developed by Bala et al. [1]. DynamoRIO [5, 6, 14] 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. DynamoRIO is also used in a project that includes a manager for bounded caches that clears no longer needed blocks out of the code cache [14]. The main purpose of the DynamoRIO project is shifting from a binary optimizer to a binary instrumentation system.

Compared to fastBT, DynamoRIO translates machine code into an IR and uses a binary optimization framework and heavy-weight transformations to generate code. fastBT on the other hand uses a light-weight table based approach.

6.3 PIN

PIN [18] is a dynamic instrumentation system that exports a high-level instrumentation API that can be used during the runtime of a program. The system offers an online high-level interface to the instruction-flow of the instrumented program. A user program can define injections or alterations of the instruction stream of a running application. PIN uses the user supplied definition and dynamically instruments the running program. PIN employs code transformations like register reallocation, inlining and instruction scheduling to make the instrumented code fast, but PIN still must pay for the high-level interface. Unfortunately the instrumentation system of PIN is distributed in binary only, except the library, which is available as open source.

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.

6.4 Valgrind

Valgrind is described in [20] as heavy-weight dynamic binary analysis tool. The main applications are checkers and profilers. Machine code is translated into an IR, which is then instrumented using high-level plugins. Valgrind keeps information about every instruction and memory cell that is used at runtime. This feature makes it possible to implement new and different instrumentation tools that are not possible in other environments. Valgrind is designed primarily for flexibility and not for performance. This makes Valgrind perfect for basic block profiling, memory checking and debugging. But due to the high overhead Valgrind is unsuited for large and performance critical programs as shown by experiments with the SPEC CPU2006 benchmark suite.

7. Conclusion

Dynamic binary translation is an important building block of many software systems, and its importance is likely to increase in the future. We use fastBT to gain insight into the overheads of binary translation. fastBT is a low overhead, simple, table-based binary translator that yields good performance: the translation is fast and incremental, and the translated SPEC CPU2006 programs execute with low overhead of 6% on average and between -3% and 6% overhead for the majority (17 out of 28) of programs.

fastBT demonstrates that a high-level interface to the binary translator is not an impediment to low overhead in the translated program. The lean translation process makes it possible to use binary translation also in scenarios that include short running programs. Other binary translators that build up expensive internal datastructures have only a slight edge for long running programs but are unable to recover the substantial translation overhead for short running programs. As we envision the use of binary translators for a wide range of scenarios (e.g., sandboxing, software transactional memory) that involve programs with short execution time, dynamic binary translators must limit the translation overhead. Only then it is possible to obtain efficiency in the execution of the translated program as the cost of translation incurs an execution penalty in the context of a dynamic translator.

Different forms of indirect control transfers limit performance of software-based translations. fastBT lowers the overhead of all indirect control transfers using self-modifying optimizations that select a suitable configuration for each individual location at runtime instead of a global selection of a single optimization for all indirect control transfers.

Using the table generator and a high-level interface a user can configure the binary translator at compile time and obtain a more flexible design than is possible by using low-level tables.

Binary translation is 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 large-scale 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* (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.

[12] GARG, M. Sysenter based system call mechanism in linux 2.6 (http://manugarg.googlepages.com/-systemcallinlinux2_6.html).

Source code of the fastBT binary translator may be downloaded at *http://www.lst.inf.ethz.ch/research/fastBT/*

[13] GILL, S. The diagnosis of mistakes in programmes on the edsac. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences 206*, 1087 (1951), 538–554.

[14] HAZELWOOD, K., AND SMITH, M. D. Managing bounded code caches in dynamic binary optimization systems. *TACO '06 3*, 3 (2006), 263–294.

[15] 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).

[16] 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.

[17] KISTLER, T., AND FRANZ, M. Continuous program optimization: Design and evaluation. *IEEE Trans. Comput. 50*, 6 (2001), 549–566.

[18] 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.

[19] 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.

[20] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07* (New York, NY, USA, 2007), pp. 89–100.

[21] 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.

[22] PAYER, M., AND GROSS, T. Requirements for fast binary translation. In *2nd Workshop on Architectural and Microarchitectural Support for Binary Translation* (2009).

[23] SRIDHAR, S., SHAPIRO, J. S., AND BUNGALE, P. P. HDTrans: a low-overhead dynamic translator. *SIGARCH Comput. Archit. News 35*, 1 (2007), 135–140.

[24] 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.

[25] 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).