Hybrid Binary Rewriting for Memory Access Instrumentation

Amitabha Roy
University of Cambridge
amitabha.roy@cl.cam.ac.uk

Steven Hand
University of Cambridge
steven.hand@cl.cam.ac.uk

Tim Harris
Microsoft Research, Cambridge
tharris@microsoft.com

Abstract

Memory access instrumentation is fundamental to many applications such as software transactional memory systems, profiling tools and race detectors. We examine the problem of efficiently instrumenting memory accesses in x86 machine code to support software transactional memory and profiling. We aim to automatically instrument all shared memory accesses in critical sections of x86 *binaries*, while achieving overhead close to that obtained when performing manual instrumentation at the source code level.

The two primary options in building such an instrumentation system are static and dynamic binary rewriting: the former instruments binaries at link time before execution, while the latter binary rewriting instruments binaries at runtime. Static binary rewriting offers extremely low overhead but is hampered by the limits of static analysis. Dynamic binary rewriting is able to use runtime information but typically incurs higher overhead. This paper proposes an alternative: hybrid binary rewriting. Hybrid binary rewriting is built around the idea of a persistent instrumentation cache (PIC) that is associated with a binary and contains instrumented code from it. It supports two execution modes when using instrumentation: active and passive modes. In the active execution mode, a dynamic binary rewriting engine (PIN) is used to intercept execution, and generate instrumentation into the PIC, which is an on-disk file. This execution mode can take full advantage of runtime information. Later, passive execution can be used where instrumented code is executed out of the PIC. This allows us to attain overheads similar to those incurred with static binary rewriting.

This instrumentation methodology enables a variety of static and dynamic techniques to be applied. For example, in passive mode, execution occurs directly from the original executable save for regions that require instrumentation. This has allowed us to build a low-overhead transactional memory profiler. We also demonstrate how we can use the combination of static and dynamic techniques to eliminate instrumentation for accesses to locations that are thread-private.

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

General Terms Design, Performance, Algorithms

Keywords Binary Rewriting, Transactional Memory

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.
VEE'11, March 9–11, 2011, Newport Beach, California, USA.
Copyright © 2011 ACM 978-1-4503-0501-3/11/03...$10.00

1. Introduction

The recent shift towards multicores has led to a large body of research that deals with shared memory multithreaded applications. The focus areas range across improved safety through race detection [14], to profiling [22], to improved scalability using software transactional memory [12]. In each of these cases, researchers have used runtime methods leveraging existing dynamic binary rewriting engines for instrumentation. Two of these applications – software transactional memory and profiling – form the motivation for the x86 binary instrumentation system in this paper.

Dynamic binary rewriting is attractive for these applications since it does not require source code availability or modification. Unfortunately, dynamic binary rewriting traditionally incurs large overheads. An alternative, that also operates at machine code level is static binary rewriting. Static binary rewriting does not suffer from the runtime overheads of dynamic binary rewriting. Unfortunately, static binary rewriting achieves only limited insight into executed code paths; for example it is difficult to determine the targets of indirect branches in binaries with static techniques alone. This in turn limits the effectiveness of static analysis in determining instrumentation points or optimisations.

In this paper, we describe an instrumentation infrastructure that combines some of the best ideas from static and dynamic binary rewriting into an instrumentation technique we call *hybrid binary rewriting*. Hybrid binary rewriting generates instrumentation at runtime. However, instead of discarding generated instrumentation at the end of execution, it is placed in an on-disk file called the persistent instrumentation cache (PIC). This "active" mode of instrumented execution discovers and instruments code as it is run, thereby providing all the benefits of dynamic binary rewriting. In addition to "active" mode, our instrumentation system also allows execution in "passive" mode. In this mode execution proceeds out of the native binary unless an instrumented version is available in the PIC. If so, it executes the instrumented version. If the PIC contains all the necessary instrumentation then "passive" mode approximates the low overhead that is obtained from the static pre-instrumentation of binaries.

The instrumentation system in this paper automatically instruments shared memory accesses in critical sections — i.e. we only wish to instrument code in critical sections delimited by lock acquire and release calls. For example, consider the fragment of source code in Figure 1. It shows a portion of code from the SSCA2 benchmark in the STAMP suite [5]. We are interested in the critical section delimited by the `TM_BEGIN()` and `TM_END()` calls. There are two shared memory accesses in that region that have been annotated using `TM_SHARED_READ` and `TM_SHARED_WRITE` calls. The STAMP benchmark already contains instrumentation for shared memory accesses within transactions (critical sections protected by a single global lock). Clearly, inserting such instrumentation is cumbersome and error prone. Our instrumentation infrastructure automatically

```
void
computeGraph (void* argPtr)
{
  ...
  ULONGINT_T j;
  ULONGINT_T maxNumVertices = 0;
  ULONGINT_T numEdgesPlaced = SDGdataPtr->numEdgesPlaced;
  ...
  TM_BEGIN();
    long tmp_maxNumVertices =
      (long)TM_SHARED_READ(global_maxNumVertices);
    long new_maxNumVertices =
      MAX(tmp_maxNumVertices, maxNumVertices) + 1;
    TM_SHARED_WRITE(global_maxNumVertices, new_maxNumVertices);
  TM_END();
  ...
```

Figure 1. Annotated fragment from the SSCA2 benchmark

places the same instrumentation at very little extra overhead (on average 26%).

The rest of this paper is organised as follows. In Section 2 we discuss the relevant merits of the two binary rewriting approaches. Next, in Section 3 we discuss how hybrid binary rewriting works, in particular the difference between running in active and passive instrumentation modes. We then cover the operation of active instrumentation mode (§3.1) and the operation of passive instrumentation mode (§3.2). In Section 4 we focus on two applications for the instrumentation system: a profiler for critical sections (§4.1) that combines memory access traces with lock contention data to produce useful transactional memory related profiles for the binary; and a system for automatically eliding locks in x86 binaries to execute critical sections using software transactional memory (§4.2). Finally, we focus on how hybrid binary rewriting can enable interesting instrumentation features that are normally not possible with either static or dynamic binary rewriting alone. The first is a technique to dramatically increase the rate at which the PIC reaches completion i.e. to ensure it contains instrumented versions of all basic blocks in all critical sections (Section 5). The second is a technique to automatically filter out thread-private locations in the instrumentation (Section 6). This is critical to approaching the performance of manual instrumentation, where the programmer is aware of – and exploits – the fact that locations that are thread private need not be instrumented when using software transactional memory.

2. Binary Rewriting Approaches

Research in the area of instrumenting machine code has been driven both by the need to build profiling tools that operate at the binary level (such as [16]), as well as for tools that actively modify execution by eliminating dead code [3] or even automatically applying software transactional memory [12]. Given x86 machine code contained in a program, there are two prevalent approaches to rewrite binary code into an instrumented form: one is purely static, while the other is purely dynamic. We next discuss each of these approaches, focusing on their strengths and weaknesses with regard to our intended applications.

2.1 Static Binary Rewriting

Static binary rewriting modifies binaries before execution to produce an instrumented version. An early example of static binary rewriting is the binary rewriting tool Atom [8]. Other examples are Diablo [21] and PLTO [17]. Static binary rewriting operates by reading an executable file (or object file), disassembling it, and rewriting instructions as desired (e.g. to insert profiling code).

```
if(AnalysisOpaqueCondition()) {
  pthread_mutex_lock(&lock);
}
pthread_mutex_lock(&possibly_nested_lock);
...
pthread_mutex_unlock(&possibly_nested_lock);
// Should the following be instrumented ?
...
if(AnalysisOpaqueCondition()) {
  pthread_mutex_unlock(&lock);
}
```

Figure 2. Possibly Nested Locking

Static binary rewriting has one major advantage over dynamic binary rewriting: there is no runtime overhead incurred to insert the instrumentation, since the process occurs before the binary is executed. However, from the perspective of our intended applications there are two key difficulties with using static binary rewriting.

The first problem arises due to indirect branches. Static binary rewriting needs to analyse the control flow graph to decide which basic blocks[1] in the binary need to be instrumented. For example, in the case of software transactional memory, the critical section comprises all basic blocks reachable from the basic block containing the lock call, but without encountering an unlock call. This cannot, in general, be determined *a priori* with static binary rewriting[2].

The second problem is demarcating critical sections in the presence of nested locking. Consider the example code fragment shown in Figure 2. A purely static approach cannot determine whether the portion of code after the first unlock call should be instrumented since its inclusion in a critical section depends on a dynamically evaluated condition. Runtime information is critical to being able to make such decisions correctly.

2.2 Dynamic Binary Rewriting

Dynamic binary rewriting modifies binaries at execution time to insert instrumentation. Dynamic binary rewriting has gained popularity since it enables extremely useful program analysis and optimisation tools to be built. A number of dynamic binary rewriting engines have been built such as PIN [10], FastBT [13], Dynamo [2] and Valgrind [11]. They have formed the basis for useful program analysis tools such as Memcheck [18], and for program optimisation, for example using Dynamo. Since instrumentation is inserted dynamically, it does not suffer from the limitations of static rewriting mentioned above. However, dynamic binary rewriting can suffer from high overhead. There are two primary sources of this overhead.

The first is the cost of inserting instrumentation. Code execution must be stopped in order to rewrite it with instrumentation inserted. This happens every time new instrumentation is inserted. This cost is particularly high for short programs, or for those with little locality.

Another source of high overhead is maintenance of the *code cache*. Since the dynamic binary rewriting engine cannot at any point guarantee that no new code requiring instrumentation will be executed, it executes all code out of a code cache. The code cache contains all encountered basic blocks, even those which have not been instrumented; this ensures that the dynamic binary rewriting engine

[1] A single-entry single-exit sequence of machine code

[2] Atom used a "grey box" approach of understanding the manner in which `case` statements are complied (by a C complier) to work out possible targets of indirect branches; however such approaches are fragile in the face of language or compiler changes.

Benchmark	Description	Basic Blocks Executed (static)	
		Overall	*Within CS*
Bayes	Learn a Bayesian network	5641	763 (13.6%)
Genome	Gene Sequencing	4243	220 (5.2%)
Intruder	Intrusion Detection	4556	476 (10.4%)
Kmeans	Clustering	4902	101 (2.1%)
Labyrinth	Routing in a maze	4894	342 (7.0%)
SSCA2	Efficient graph representation	4630	105 (2.3%)
Vacation	Scaled down SpecJBB	4866	622 (12.8%)
Yada	Delaunay mesh refinement	6403	1006 (15.7%)

Table 1. STAMP: Basic Block Profile from Execution

maintains control of code execution and is able to see all newly executed code. Unfortunately the code cache imposes significant overhead even for un-instrumented code. This stems from the cost of maintaining the finitely sized cache, and hence taking care of events such as evictions, as well as linkage between basic blocks.

From our perspective, the code cache is a completely unnecessary source of overhead. There is no need to instrument code outside critical sections, and thus no need to put it into the code cache, possibly displacing more useful instrumented blocks. This is illustrated in the profile of basic blocks from the STAMP benchmark suite shown in Table 1. For each benchmark, this shows the number of (static) blocks in the binary that were actually executed while the next column shows the number of those that were within a critical section (CS). It is clear that only a small fraction of basic blocks need be executed with instrumentation, and thus it is desirable to avoid any overhead (in terms of both code cache space and execution time) for the others.

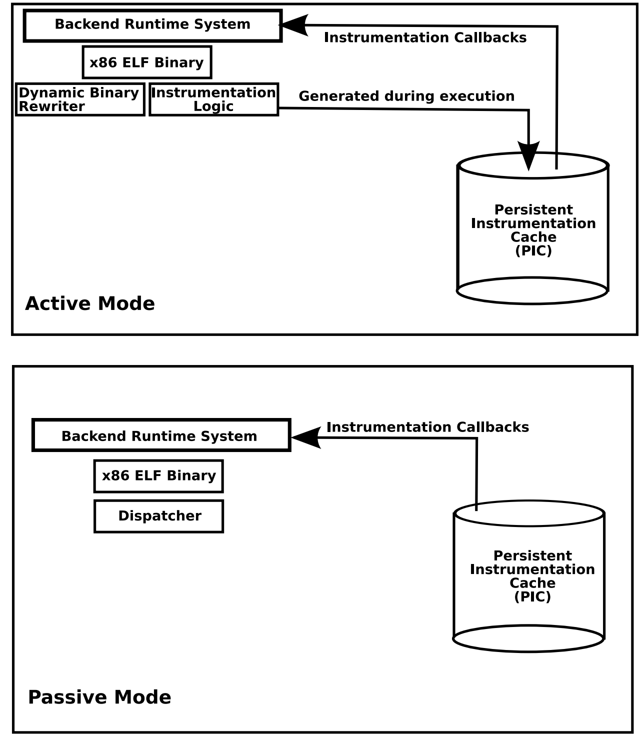

Figure 3. Instrumentation Flow

3. Hybrid Binary Instrumentation

The x86 binary instrumentation system in this paper aims to combine the benefits of static and dynamic binary rewriting while sidestepping their problems. At the heart of this hybrid instrumentation system lies the Persistent Instrumentation Cache (PIC).

A PIC contains instrumented versions of basic blocks within critical sections of its originating binary. It is persistent, i.e. held in an on-disk file. In form it thus resembles instrumentation that would have been added by a static binary rewriting engine. A *complete* PIC contains instrumented versions of *every* reachable basic block within *every* possible critical section of the binary. The completeness of a PIC is clearly undecidable in the presence of indirect branches in the binary. We return to the problem of tolerating incomplete PICs later in the paper.

The PIC is generated dynamically, and we depend on execution to look past indirect branches. We also depend on execution to properly handle nested critical sections (such as the example in Figure 2) by dynamically counting held locks. The PIC is thus generated by operating in a dynamic instrumentation mode.

The process of generation of the PIC is shown in Figure 3. The "Backend Runtime System" consumes instrumentation and contains callbacks for the instrumentation hooks. The instrumentation flow starts with a number of iterations of *active* execution (top half of figure). In this mode, we depend on a dynamic binary rewriting engine to intercept all executed code in the binary. Basic blocks in critical sections are instrumented and placed in the PIC.

Once the PIC obtains sufficient coverage, we can execute in *passive* mode (bottom half of the figure). In this mode a low cost "dispatcher" loads the PIC into memory and intercepts lock calls. Within a critical section, instrumented versions of basic blocks are executed *if available*. Passive mode provides best-effort instrumentation but cannot itself add new instrumentation to the PIC. If the PIC is incomplete and execution enters a basic block that has no instrumented version in the PIC, a special instrumentation hook informs the consumer of the instrumentation of this event in order that it may take appropriate action. Execution then switches to the *un-instrumented* version of the critical section.

We now describe in detail the operation of each of these modes of instrumentation.

3.1 Active Instrumentation Mode

Active mode uses a dynamic binary rewriting engine to intercept execution and instrument basic blocks in critical sections. Dynamic binary rewriting engines are fairly complex to build and maintain. We thus chose to leverage an existing dynamic binary rewriting engine for this part of the instrumentation system. We use PIN [10], a widely used and stable dynamic binary rewriting engine for x86 binaries. Our decision to use PIN was guided by two factors.

First, although the source code for the core modules of PIN are not publicly available, it provides a high level API through which it can be controlled and extended. The user provides a "pintool" written in C++ that uses this API to inspect and manipulate x86 code. The API can operate at various levels of abstraction: from whole images, down to functions, basic blocks and individual instructions. PIN also includes a lower level (and not so widely used) API to directly decode, manipulate and re-encode x86 instructions (complex due to their CISC nature) from and to machine code called the X86 Encoder Decoder (XED). We made extensive use of XED to build the instrumentation system in this paper.

Second, PIN has a large community of users and is actively maintained. This is important because the x86 ISA is actively changing

```
...
strcpy(dst1, src1, size1);
...
// enter critical section
pthread_mutex_lock(&lock);
  ...
  strcpy(dst2, src2, size2);
  ...
pthread_mutex_unlock(&lock);
```

Figure 4. The same function in multiple contexts

(e.g. the addition of SSE4 instructions), and it is important that the binary rewriting engine keep up with these additions to be useful for our applications now and in the future.

3.1.1 Critical Sections

There are two subtle problems in determining critical section boundaries and intercepting all executed code within them.

The first is that of accurately delimiting critical sections (a problem already mentioned in §2.1). We require the backend runtime system to maintain per-thread counters to determine when critical sections are begun, and when they end. On a critical section begin, we require a call to a specially named function in the backend runtime (`cs_begin`); similarly, on encountering the end of a critical section, we require a call to another specially named function (`cs_end`). The instrumentation infrastructure looks for execution of these functions (which can be empty "no-ops") in order to learn when critical sections begin and end.

The second problem is ensuring that any code that is executed both within and outside critical sections is properly instrumented. Consider Figure 4. The string copy function is first called outside critical section context. We thus do not generate an instrumented version. Later it is called within a critical section context. This time, however, PIN does not present us with the basic blocks in `strcpy` since it has already added them to its code cache (unmodified since we did not see them in critical section context).

PIN allows basic blocks to be annotated with a version that can be propagated through branches out of a basic block to their target basic blocks. The same basic block with a different version is treated differently and presented individually for instrumentation. We hence annotate basic blocks within a critical section with a special tag, ensuring that, for example, the `strcpy` function in Figure 4 is presented again for instrumentation on the subsequent execution because it has a different tag.

3.1.2 Instrumentation

We generate instrumentation for critical section begin/end calls and for each shared memory access within a critical section. The instrumentation for critical section begin/end calls are tailored to provide a variety of information to the backend runtime system. For our STM application, that replaces locks with transactions in the binary, the region begin call provides information such as the actual lock acquired, the type of lock (pthreads mutex, openmp nested lock, etc.); information about the current top of stack (before the call to the instrumentation routine); and the size of the function frame where the lock call is encountered. The last two pieces of information are used by the STM runtime system to construct a precise checkpoint (together with a `setjmp` call) that can be used to rollback execution in the event that it encounters a conflict (through a `longjmp` call followed by a copy to restore the stack frame). The instrumentation can thus be tailored to suit the particular backend runtime system being used.

The second type of instrumentation generated is for shared memory accesses within a region. Figure 5 shows the example of a basic block in a shared memory region from one of the benchmarks we used. The numbered instructions on the right correspond to the numbered instructions on the left. For example, the first instruction accesses memory. This is converted into an instruction that first loads the target address into the `eax` register. The next few instruction load the size of the access into the `edx` register, and set a flag – stored in `ecx` – which indicates whether or not this instruction is a read-modify-write (both a read and a write) instruction. The size and read-modify-write flag are encoded such that the most common values (4 bytes, false) map to zero. This means the registers can be set up with a two byte instruction (exclusive or-ing the register with itself), keeping the size of instrumentation and hence instruction cache pressure down. The call to the instrumentation hook returns the (possibly different) address to use for the memory access in `eax`, which is then used in the instrumented version of the basic block.

The first notable feature of the instrumentation is CPU flag and register management. Since the call to the instrumentation hooks is expected to destroy the `eax`, `edx` and `ecx` registers as well as the flags, these need to be saved and restored as appropriate. This is accomplished by the un-numbered instructions in the instrumented version of the basic block. The save area is setup on stack (the PIC is shared between threads) by the first instruction. We perform liveness analysis at the level of the basic block to optimise away unnecessary save restores — for example, the fourth instruction overwrites the x86 flags and thus the flags are not saved.

The second notable feature of the instrumentation is the treatment of memory accessed through the stack pointer (register `esp`). The stack is usually thread private and (due to the limited number of registers on the x86) heavily accessed. Assuming accesses to the stack to be thread private means that they can be performed directly in our applications. However, stack accesses need to be adjusted to account for the save area we create on stack. In the example this can be observed in instruction 5, where the offset is adjusted upward by 16 bytes.

The final notable feature about the instrumentation is the handling of the `call` instruction that terminates the basic block. The instrumented version pushes the return address before jumping to the target. This is standard practise for binary rewriting engines and originates from the need to leave return addresses unmodified on stack. In the example, the `rebalance_insert` function would see the original native address rather than the address from the PIC were it to query the return address of the function. A common occurrence of this kind of behaviour is in position independent code, where a call is made to the immediately following instruction, which then queries the top of stack to discover the current instruction pointer. This is done as there is no direct way on 32-bit x86 to materialise the instruction pointer in a general purpose register.

Memory access instrumentation is also complicated by the fact that the x86 Instruction Set Architecture permits complex instructions. Some instructions allow accessing more than one location (such as a `push` of the contents of a memory location). Another complication arises from string operations where the length of the access cannot be determined statically (it usually depends on the contents of the `ecx` register). We handle such cases by breaking them down into simpler RISC style operations that are then instrumented.

3.1.3 PIC Operations

There are four basic operations performed on the PIC in active mode. These are *(i)* loading the PIC into memory; *(ii)* appending

```
// Note: AT&T format-> operation src, dst
1. subl   $0x1,0x8(%eax) # Memory[8 + Reg[eax]] -= 1;
2. mov    %esi,0xc(%eax) # Memory[12 + Reg[eax]] = Reg[esi];
3. mov    %eax,0x4(%esp) # Memory[4 + Reg[esp]] = Reg[eax];
4. xor    %ebx,%ebx # Reg[ebx] = 0;
5. mov    %edi,(%esp) # Memory[Reg[esp]] = Reg[edi];
6. call   8048ba0 <rebalance_insert>
```

```
    lea    0xfffffff0(%esp),%esp
    mov    %eax,0x0(%esp)
1.1 lea    0x8(%eax),%eax
    mov    %ecx,0x4(%esp)
    mov    %edx,0x8(%esp)
1.2 xor    %edx,%edx
1.3 xor    %ecx,%ecx
1.4 inc    %ecx
1.5 call   0xff6a4730 # Instrumentation
1.6 subl   $0x1,(%eax)
    mov    0x0(%esp),%eax
2.1 lea    0xc(%eax),%eax
2.2 xor    %edx,%edx
2.3 xor    %ecx,%ecx
2.4 call   0xff6a4730
2.5 mov    %esi,(%eax)
    mov    0x0(%esp),%eax
3.  mov    %eax,0x14(%esp)
4.  xor    %ebx,%ebx
5.  mov    %edi,0x10(%esp)
    mov    0x8(%esp),%edx
    mov    0x4(%esp),%ecx
    lea    0x10(%esp),%esp
6.1 push   $0x8048e12
6.2 jmp    0x123c4ba0
```

Figure 5. Shared memory instrumentation for a basic block

instrumented basic blocks to the PIC; *(iii)* executing from the PIC; and finally *(iv)* querying the PIC.

We load the PIC into memory by doing a memory map (Unix `mmap`) from the disk file containing it. This ensures that the disk file is up-to-date with any additions to the PIC. Appending basic blocks to the PIC simply consists of writing out instrumented versions of basic blocks to the end of the PIC.

Executing from the PIC presents a problem due to the special handling of self-modifying code implemented by PIN. In order to detect self-modifying code, PIN looks for pages that are being executed from while being marked writable. It then marks these pages as read-only and traps writes to them in order to detect any self-modifying code. This causes large slowdowns when executing instrumented code out of PIC pages. To work around this problem, we map the same PIC page twice, once as executable but read-only and once as read-write but not executable. Appending to the PIC is done through the writable mapping while actual execution uses the executable read-only mapping.

The final operation that needs to be supported by the PIC is queries to map executable native addresses to instrumented basic block addresses in the PIC, if present. The core of the logic that handles queries is a map:

$$f : \text{native address} \rightarrow \text{PIC offset}$$

Such a map is easy to setup and maintain for a single run but difficult to persist across runs. The reason is that the native executable address originating the instrumented basic block in the PIC can change across runs. For example, the native address might originate in a shared library that can change its load address on each active execution. To solve this problem, the map is persisted as:

$$f : (\text{native address relative to base, image name}) \rightarrow \text{PIC offset}$$

It is loaded and turned into the required form by querying the base of each loaded image (main binary or shared library). A similar technique is used by dynamic binary rewriting engines that persist instrumentation across runs [15].

3.2 Passive Instrumentation Mode

We now discuss instrumented execution in passive mode. In this mode, we use the Unix `LD_PRELOAD` mechanism to accompany the x86 binary with the instrumentation system and the backend, both of which are implemented as shared libraries. The critical feature of execution in passive mode is simplicity and low overhead. There is zero overhead to add instrumentation and, as we shortly show, zero overhead when executing code outside an instrumented critical section.

3.2.1 Preparation

As Figure 3 shows, the PIC prepared during active instrumentation can be used in passive mode. An offline tool needs to be run on the PIC before any passive execution that follows an active execution. The job of this offline tool is effectively to "stitch" together basic blocks in the PIC by patching branches across them, to target instrumented basic blocks in the PIC rather than in the native binary. As an example, consider the call instruction at the end of Figure 5. During active execution it targets the native binary and is intercepted and redirected via PIN. The offline patching step patches the branch to point to the instrumented version of the target. Note that on the x86, direct branches are instruction pointer relative and thus the patching is unaffected by PIC relocation across different runs. The patching step is fast — for example, it takes barely a few seconds for a 5MB PIC.

3.2.2 Intercept and Dispatch

The heart of passive execution is the intercept and dispatch logic. The first step is to intercept all lock calls. This is done through the unix `LD_PRELOAD` mechanism by the intercept logic, and is specific to the type and functionality of the locking in use. When a lock call is intercepted, any instrumentation hooks related to lock acquisition are invoked, and then control is transferred to the dispatcher.

The dispatcher queries the PIC to determine the instrumented version of the basic block pointed to by the return address of the original lock call. It then *modifies* the return address on the stack to

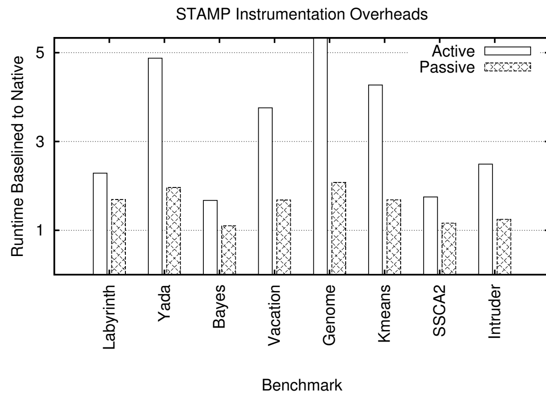

Figure 6. Instrumentation Overhead

point to the instrumented version of the basic block (the PIC having been mapped into memory). On exit from the dispatcher, control transfers into the PIC and executes the instrumented version of the critical section. The unlock call is replaced with a call into the instrumentation hook, which indicates on return if any locks are held. If no locks are held, control returns to the native binary. Otherwise control returns to the dispatcher which decides the appropriate basic block to branch to in the PIC.

The result of this scheme is that, when executing in passive mode, there is no overhead for inserting instrumentation (as it has already been inserted during the active phase). Nor is there any overhead when executing *un-instrumented code* outside any critical section, as in such cases execution proceeds directly from the native binary.

The dispatcher is also used to resolve indirect branches by looking up the PIC. Finally, if a query into the PIC fails due to the PIC not being complete, an exception is raised to the backend. The default behaviour on an unhandled exception is to switch to executing un-instrumented code out of the native binary.

3.2.3 Passive Instrumentation Overheads

In this section, we evaluate the difference in overhead between running in active and passive modes. We use a specially constructed "no-op" backend in this section, that simply acquires and releases necessary locks at critical section boundaries (substituting the original lock and unlock calls) and directly returns the address passed in for shared memory references. The only overheads left are thus the instrumentation call overhead, and that due to PIN (in active mode) or the dispatcher (in passive mode). Other backends illustrating various case studies follow later in the paper.

We use the STAMP benchmarks to demonstrate how the dispatcher lowers overheads. We use a single global lock to implement the transactions in STAMP and thus the `TM_BEGIN` and `TM_END` calls (such as in Figure 1) are compiled to pthread lock acquire and release calls. We use macros to turn the manual instrumentation of shared memory accesses (such as shown in Figure 1) into direct memory accesses. We then use the instrumentation system presented thus far to instrument these automatically in the binary. Our baseline is the x86 binary running *without any instrumentation*.

Figure 6 shows the performance overhead with this no-op backend for the STAMP benchmarks (running with 16 threads). We measure the execution time running with instrumentation baselined to (divided by) the execution time of the binary running without instrumentation. Using the dispatcher instead of PIN is faster in all cases, with benefits ranging from 26% for labyrinth to as much as 61% in the case of genome.

A key limitation of the dispatcher is that it depends on critical sections being delimited by shared library calls (in order to use the `LD_PRELOAD` mechanism). In future work, we intend to remove this restriction by providing a means to redirect function execution within the loaded binary. This can be done, for example, by placing an appropriate branch instruction at the first few bytes of the function after the binary is loaded into memory.

4. Case Studies

In the following we describe two case studies (backend runtime systems) that make use of the PIC through more complex backends We demonstrate both of them using STAMP, starting with a pre-built PIC, running in passive mode. Note that we use the same binaries and the *same PIC* for both the case studies, since the backend is decoupled from the instrumentation.

4.1 STM Profiler

The profiler uses ideas from a similar profiler we have built earlier to predict transactional memory performance [16]. That work used PIN to trace memory accesses in a critical section as well as to measure lock contention. Unfortunately PIN added significant overhead, making it extremely difficult to measure the time spent waiting for a lock accurately. This instrumentation system allows a simple remedy to that problem by using passive execution mode.

An example output from the profiler for the vacation benchmark of STAMP is shown in Figure 7. For each critical section (source line number is optionally obtained from debug information for the binary), the profiling tool prints the fraction of total execution time spent waiting for and executing the critical section followed by the average number of waiters seen for the lock. It then prints properties of the critical section: number of shared memory reads and writes instrumented, the number of locations reads from and written to and, finally, the *dependence density*. This last metric [22] is the probability that, were the critical sections scheduled in parallel, there would be a data flow dependence seen by a dynamic instance of this critical section. It essence it estimates the conflict probability were the binary to be run with transactional memory.

The crucial point about the profiler output is that the first two metrics depend on accurate timing information about locks and minimum instrumentation overhead. The last five metrics depend on tracing all memory accesses in a critical section (the log files are post-processed later) and thus impose significant overhead. In order to satisfy both these goals in a single run of the binary, the dispatcher for profiling implements four phases of execution, shown in the state machine of Figure 8. Lock timing and waiters-related information is collected in the `timing` phase. Critical section tracing is done in the `tracing` phases. No information is collected during the silent phases. Tuning the length of the four phases changes the sampling rate (fraction of critical sections instrumented for either timing or tracing). The dispatcher switches to the instrumented version of the critical section in the PIC only in the `tracing` phase, thus eliminating any tracing overhead when it is not needed. This flexibility in applying instrumentation is essential for building an efficient and accurate profiler.

Finally, the profiler ignores exceptions raised in passive instrumentation mode, since un-instrumented execution in a critical section only affects the accuracy of the profiled data and does not affect correct execution of the binary.

CS	cs_frac(%)	wait_frac(%)	avg_waiters	rd_ops	rd_locs	wr_ops	wr_locs	dep_dens
client.c:247	0.120	0.811	6.870	447.585	203.204	20.510	14.076	0.440
client.c:267	0.041	0.882	6.868	126.768	74.950	4.363	4.307	0.016
client.c:196	8.037	82.849	6.871	447.412	127.811	12.094	11.601	0.007

Figure 7. Profiling the vacation benchmark in STAMP

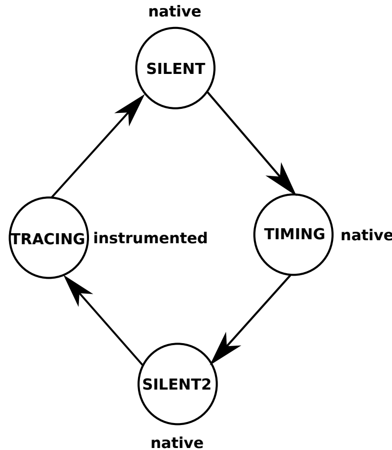

Figure 8. Profiler Phases

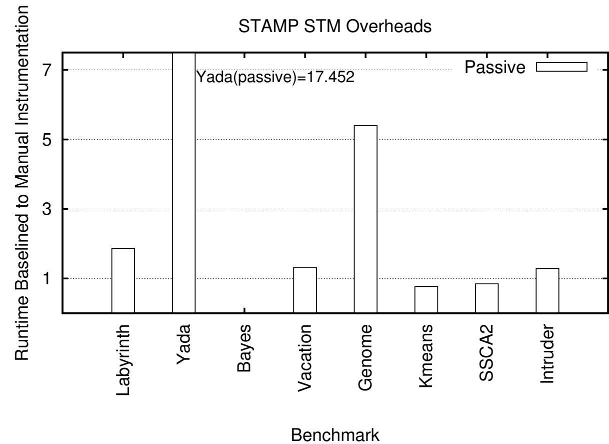

Figure 9. Instrumentation Overhead over Manual

4.2 Software Lock Elision

The second case study we discuss is software lock elision: the automatic transformation of a lock-based binary into one which uses software transactional memory. The instrumentation for shared memory accesses allows indirection of reads and writes to STM buffers. We apply the well known two-phase commit protocol that most STMs use [6] to atomically apply changes to shared memory at the end of the critical section. In order to judge the efficiency of our automatic shared memory access instrumentation, we compared against manually inserted instrumentation at the source code level. The STAMP benchmarks are already available with this instrumentation. We kept the backend runtime system (the STM implementation) the same. Figure 9 shows the overhead (running time with automatic instrumentation divided by running time with manual instrumentation) of the automatic instrumentation over the manual one, when running with 16 threads. The goal is to achieve overheads equal to that of the manual instrumentation using a purely automatic technique. Most of the benchmarks are at a ratio close to 1.0, meaning our automatic instrumentation is as efficient as the manual one.

However, Yada and Genome however show extremely high overheads (17X, 5X), and in the case of Bayes the amount of instrumented accesses is so large that it overflows the STM buffers. When we compared the data provided by the profiler (previous section) with statistics obtained from the manual instrumentation, we realised that the instrumentation infrastructure inserted far more instrumentation calls than the manual one for these three benchmarks.

To understand why this is the case, consider Figure 10, which shows a fragment of code from the genome benchmark. It includes a transaction, that subsequently makes a call to insert an entry into a hash table that uses the hash function at the bottom. The hash function includes no (manual) instrumentation of the string being hashed. This reflects the knowledge of the programmer that, although the string is shared between threads, the organisation of the program is such that past the initialisation point the string is shared read-only. A program-specific optimisation has thus been made by the programmer to remove any instrumentation in the `hash_sdbm` function. Unfortunately our instrumentation infrastructure cannot incorporate such knowledge and ends up adding a far larger number of instrumentation calls. In Section 6 we show how a combination of static and runtime techniques can be used to *safely* incorporate these optimisations automatically.

```
TM_BEGIN(); { // stamp/genome/sequencer.c:290
 long ii_stop = MIN(i_stop, (i+CHUNK_STEP1));
 for (long ii = i; ii < ii_stop; ii++) {
  void* segment = vector_at(segmentsContentsPtr, ii);
  TMHASHTABLE_INSERT(uniqueSegmentsPtr, segment,
                     segment);
 } /* ii */
} TM_END();

ulong_t hash_sdbm (char* str) {
  ulong_t hash = 0;
  ulong_t c;
  while ((c = *str++) != '\0') {
    hash = c + (hash << 6) + (hash << 16) - hash;
  }
  return hash;
}
```

Figure 10. A fragment of code from the genome benchmark

Finally, the STM needs to handle exceptions raised in passive mode correctly. This is done through the standard STM practise of *irrevocability* [23], where only one transaction is allowed to run at a time, and hence can operate without any instrumentation. Note that this only occurs for any critical sections not discovered when generating the PIC. In the next section we describe how we ensure good coverage of basic blocks by applying an on-demand static discovery technique.

5. Static Basic Block Discovery

Passive mode should ideally execute with every basic block reachable in a critical section instrumented and placed in the PIC. The instrumentation system presented thus far depends on PIN to discover basic blocks for us. PIN allows instrumentation of code at the granularity of a trace: a contiguous sequence of basic blocks terminated by an unconditional branch. Crucially, traces are presented for instrumentation only when they are about to be executed. This leaves the process of generating a complete PIC dependent on program inputs and, in the case of the multithreaded programs we are interested in, timing.

Deciding if the PIC is complete is undecidable in the presence of indirect branches. However, it is entirely decidable given only direct branches. Hence, we use proactive basic block discovery by *statically* traversing the control flow graph at *runtime*. This allows us to discover reachable basic blocks even before they are executed. We do this using a depth-first search of the control flow graph in the binary using Algorithm 1. The basic block at the root (the starting block in the trace that needs instrumenting) is added to the basic block stack, which is then passed to the algorithm. The traversal is terminated on either finding an indirect branch or a call that terminates a critical section.

Algorithm 1 Depth-First Search of Control Flow Graph

```
 1: while BasicBlockStack is not empty do
 2:    bb = BasicBlockStack.pop()
 3:    Instrument bb and add to persistent instrumentation cache
 4:    ins = bb.LastInstruction()
 5:    /* ins must be a branch */
 6:    if ins ends a critical section (unlock call) then
 7:       continue
 8:    else
 9:       if ins is a direct branch then
10:          bb = BasicBlockAt(ins.target())
11:          BasicBlockStack.push(bb)
12:       end if
13:       if ins is a conditional branch then
14:          /* has a basic block at fall-through */
15:          bb = BasicBlockAt(ins.next())
16:          goto line 3
17:       end if
18:    end if
19: end while
```

Using static traversal improves the rate at which the PIC approaches completion but we still depend on dynamic execution to discover starting points of critical sections, and to look past indirect branches. Another limitation in practise is PIN's capability to locate basic block boundaries. An interesting example we encountered when instrumenting the standard C library was an instruction sequence that checked the thread count and, if it was zero, jumped into a locked instruction at a point just past the lock prefix, effectively removing the overhead of the lock when there is only one thread. PIN does not make the target of the jump available in its list of instructions until actual execution discovers it. Note that the implementation of proactive basic block discovery pushes PIN's APIs into uses that were likely not envisaged by the developers.

Proactive basic block discovery borrows some of the best features from static and dynamic binary rewriting techniques. Practically, we found it extremely effective in quickly building the PIC. For example, we used this infrastructure extensively on the STAMP benchmarks.

Benchmark	Static + Dynamic			Dynamic		
	1	*2*	*3*	*1*	*2*	*3*
Bayes	1435	0	0	723	2	0
Genome	383	0	0	221	0	4
Intruder	629	0	0	452	2	4
Kmeans	178	0	0	91	4	0
Labyrinth	443	0	0	340	2	0
SSCA2	394	0	0	111	0	0
Vacation	853	0	0	464	0	0
Yada	1113	0	0	899	1	0

Table 2. The number of basic blocks added to the PIC for each successive iteration of instrumented execution. The combination of dynamic execution with static basic block discovery (*lhs*) enables much faster convergence than with just dynamic execution (*rhs*).

Benchmark	Executable(bytes)	CS Count	PIC size/Binary size
Bayes	181603	18	0.34
Genome	118334	8	0.16
Intruder	153089	6	0.20
Kmeans	52821	6	0.17
Labyrinth	116384	6	0.18
SSCA2	140156	13	0.14
Vacation	143772	6	0.28
Yada	196715	9	0.29

Table 3. Persistent Instrumentation Cache Space Costs

As Table 2 shows, with static walking of the control flow graph the PIC converges within one iteration. On the other hand, without static walking of the CFG, the PIC does not converge even after three iterations. Another advantage of proactive code discovery is that we were able to run STAMP with reduced inputs sets in order to build the PIC. Thus, even with the overhead of interception using PIN, building the PIC for all the benchmarks took only a minute and 18 seconds. On the other hand, running with the STM backend took 18 minutes. Proactive basic block discovery proved to be an invaluable time saving tool for much of our research.

In spite of proactive basic block discovery, the PIC is space efficient since it usually holds only a fraction of the actual executable, unlike a dynamic binary rewriting engine that would ultimately hold all executed code in its code cache. Table 3 shows the original executable size, the number of critical sections and the size of the fully generated PIC as a fraction of the executable size. Bayes has the largest relative size of the PIC at 34%. In reality the relative size of the PIC is even smaller since the static executable size does not take into account shared library code that may be called. In the case of Bayes, for example, the PIC includes an instrumented version of the glibc `quicksort` function.

6. Private Data Tracking

Making automatic shared memory access instrumentation practical requires some way to distinguish data that is thread-private from that which is not. This is a problem for data on the heap, as there is no clear way to distinguish these (unlike either global thread private variables – which are usually accessed through a special segment base – or auto variables on the stack).

The Genome, Bayes and Yada benchmarks in STAMP represent three cases where this is problematic, as was seen in Figure 9. One way to get around this issue is to expose an annotation that indicates thread-private variables to the instrumentation methodology. For example, some compilers that automatically instrument STAMP [20] choose to make such annotation visible to the com-

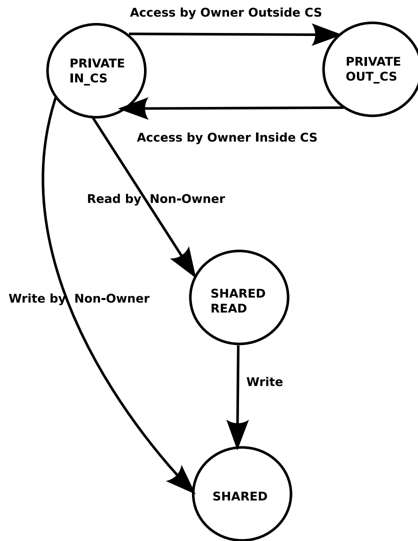

Figure 11. State machine for allocation pools

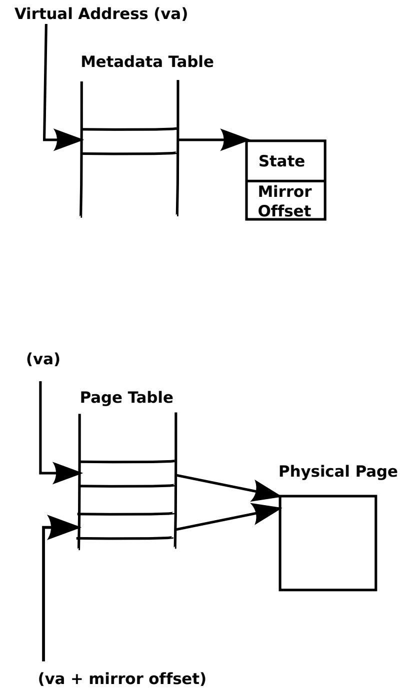

Figure 12. Memory mapping for allocation pools

piler as a language extension. This is not possible in our case since we operate at the binary level where source code information is not available. Hence we wanted to come up with an automatic technique that can reduce the impact of these extra barriers. In this section we present a private data tracking (PDT) scheme that safely reduces the cost of instrumentation for thread-private locations without using any source code information.

6.1 Allocation Pools

Our solution starts with the observation that it is usually possible to discriminate thread-private heap data from that which is not by considering their allocation sites. In the STAMP benchmarks, for example, thread private data is allocated at different physical call sites in the binary from data that is shared. However, it is not possible to statically know which allocation site allocates thread-private data. Instead we start with the assumption that all allocation sites allocate thread-private data and then dynamically detect when an allocation site has allocated data that turns out to be shared between threads.

A complication with this approach is that it is not possible to intercept memory accesses outside critical sections (access to shared data can happen outside critical sections using other methods of synchronisation). A key design cornerstone for us was to allow direct execution from the native binary for any code outside a critical section. Hence we make use of hardware memory protection to detect when memory regions are shared among threads.

Our solution intercepts all memory allocation and free calls in the binary (using the `LD_PRELOAD` mechanism) and uses a separate per-thread, per-allocation site allocation pool. The pages for each allocation pool can thus be independently protected using the standard `mprotect` call in Linux.

Figure 11 shows the states that each allocation pool can be in and the possible transitions between them. The states `PRIVATE IN_CS` and `PRIVATE OUT_CS` correspond to cases where the allocation pool (and its associated pages) are owned by a single thread (private). The state `SHARED READ` represents read-only access to the entire allocation pool. Finally the `SHARED` state represents shared read-write access to the allocation pool.

We use a different privilege protection for pages in each state. Faulting accesses trap into a fault handler that changes page state to reflect sharing (transitions in Figure 11). In the read-only state only read access is permitted to the pages. In the shared state all access is permitted to the pages.

Setting page permissions for the owned (private) states is tricky. We want to be able to detect when other threads access the page while still allowing access by the owner. Unfortunately, the page tables in Linux are shared among threads and thus one cannot expose different protections for the same page to different threads. The solution we use is to separate accesses by the owner within a critical section from those outside of any critical section.

The `PRIVATE OUT_CS` state represents the case where access is only allowed outside critical sections. In this state we use the *most permissive* page protection settings, allowing all access. This might seem strange, but only results in us being unable to detect sharing when in the `PRIVATE OUT_CS` state. On the positive side however, it means that the owning thread can freely access pages from its owned allocation pools.

The `PRIVATE IN_CS` state represents the case where access is only allowed inside critical sections. In this state *all* access to the page is withdrawn. Thus any access by a non-owning thread faults resulting in a state transition. This however means that there must remain some way for the owning thread to access these pages. We accomplish this by a *mirror* mapping (Figure 12). Each page in an allocation pool is mapped twice (from a backing file on an in-memory filesystem). In addition to the virtual address visible to the application, we map it at a known mirror offset. The mapping at the mirror offset is always accessible and is used by the owner for any accesses within the instrumentation callbacks.

The two private states capture thread-private patterns where allocations made and initialised outside a transaction are used within transactions. They also capture patterns where allocations are made, used, and discarded exclusively within transactions [7].

```
b4: nop
b5: nop
b6: nop
b7: jmp     0xf1 #patched jump
bc: mov     $0x55cf7000,%edx
c2: mov     %eax,%ecx
c4: shr     $0xc,%ecx
c7: mov     (%edx,%ecx,4),%ecx
ca: test    %ecx,%ecx
cc: je      0xdc
d2: mov     (%ecx),%edx
d4: test    %edx,%edx
d6: je      0xf8
dc: push    %eax
de: mov     $0xb8,%ecx
e4: mov     $0x35,%edx
ea: call    0xfe06abb0 #patching routine
ef: pop     %eax
f1: xor     %edx,%edx
f3: call    0xfe067400 #instrumentation callback
f8: mov     (%eax),%edx
```

Figure 13. Self-Modifying Instrumentation

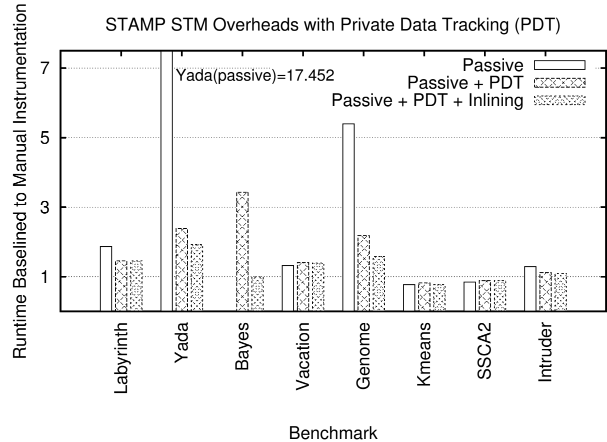

Figure 14. STAMP + Private Data Tracking

However if a thread-private allocation is used frequently both within and outside a transaction, it leads to excessive state changes with the corresponding `mprotect` calls becoming a performance bottleneck. We currently limit the occurrence of such cases by placing a threshold on the number of times an allocation pool can transition between the two private states: if this threshold is exceeded, we permanently place the pool in the shared state. In future work, we intend to explore solutions where we give each thread its own private page table, thereby simplifying the state machine and removing this case.

Finally, changing states for an allocation pool requires ensuring that no other thread can speculatively use the old state. We accomplish this through a quiescing mechanism whereby we ensure that no thread is executing out of the PIC before changing the allocation pool state.

6.2 Per-Page Metadata

In the shared read-only state, all reads simply use the original pointer for the access, since the location is shared read-only. For the `PRIVATE IN_CS` state, all reads and writes use the corresponding mirror pointer. To maintain the current state of a page, we allocate a global array whose elements are pointers for each page in the system: the metadata table in Figure 12. The entry points to a per allocation pool structure recording the current state for the pool. The instrumentation callbacks for shared memory access perform a lookup and a check to determine the correct mode of access and whether a state change needs to be performed.

Our data structures are designed to add a space overhead of under 2% to each physically accessed page. Virtual address space is also at a premium on the 32-bit machines we used for our experiments: with some benchmarks allocating as much as 1.5 GB of space, doubling the virtual address space required to 3GB for mirror maps was not feasible. Instead we make the observation that the mirror map is never required once we transition out of the two private states. We can therefore unmap the mirror space when we reach either of the non-private states.

6.3 Optimising Instrumentation

To fully realise the benefits of private data tracking, it is desirable to eliminate the call from the PIC altogether, at least for reads to thread-private memory. We do this through a combination of inlining checks within the PIC and using self-modifying code. First, we place a jump to the next instruction that begins the PDT checks. If the PDT checks succeed, we jump over the call to the instrumentation routine. On the other hand, if the PDT check fails, we call back to special patching function that patches the initial jump to the next instruction. This will directly branch to the call to the instrumentation routine, removing the inlined PDT checks.

A example of this "self-modifying instrumentation" is shown in Figure 13. The `jmp` is placed (via padding through nops) such that the branch offset is aligned to a 4 byte boundary (to avoid x86 self modifying code related quirks). The jump (in this disassembled fragment from the PIC) has been patched to jump to the callback (at `0xf3`) meaning the inlined PDT checks right afterwards must have failed during execution. If the PDT checks were to succeed, the conditional jump at `0xd6` would have moved directly to using the original pointer at `0xf8`.

This form of self modifying instrumentation works on the assumption that instructions that access thread-private locations can be statically partitioned from those that do not, and hence a single run is usually sufficient to set up the PIC appropriately to eliminate callbacks for accesses to thread-private locations. Note that if this assumption does not hold it simply means that all accesses are modified to call into the backend runtime system, which can then dynamically filter out thread-private accesses.

In Figure 14 we demonstrate the effects of private data tracking. It adds little overhead to most benchmarks but in the case of Genome and Yada it significantly reduces the overheads of automatic instrumentation. In the case of Bayes it brings the amount of shared memory access within reach of STM buffering. The average overhead for automatic instrumentation over the manual one is 26% with private data tracking and inlining enabled.

6.4 Special Case: OpenMP Thread-private Data

A special case for the PDT infrastructure is when one can statically identify allocations sites that allocate thread private data. In this case, we can relinquish memory protection while placing such data in the `PRIVATE_IN_CS` mode. Mirror maps are no longer necessary and accesses can be made directly.

An example of this is OpenMP thread private data. The OpenMP specification prohibits thread-private data from being shared between threads. Furthermore, compilers often use dynamic data al-

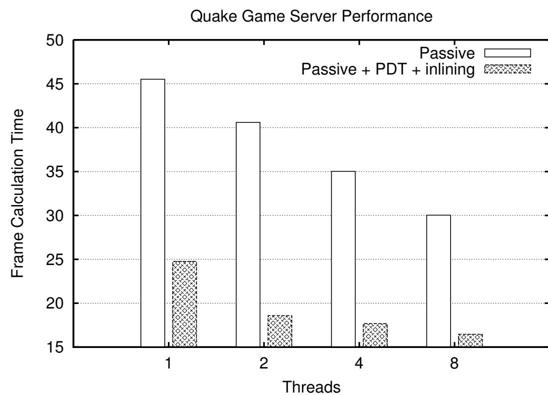

Figure 15. Quake

location for OpenMP thread-private data — for example, the Intel compilers use a specific call (`__kmpc_threadprivate_cached`) in order to allocate any thread-private data. By hooking this call we can place all such allocated thread-private data in a specially marked pool that does not require instrumentation.

We used this kind of filtering in conjunction with our instrumentation infrastructure on a multithreaded version of the Quake game server [9]. It uses OpenMP for multithreading and coarse grained OpenMP locks for synchronisation. We compiled the code using the Intel 3.0 C compiler. Figure 15 shows the benefits of filtering such accesses when applying software transactional memory to an executing game server. We report on the frame calculation time, which represents the parallel portion of the benchmark. Private data tracking significantly lowers the overhead of shared memory instrumentation for Quake.

7. Related Work

There has been significant work on instrumenting execution using dynamic binary rewriting engines in general [3, 10, 11, 19], as well as some which applies dynamic binary rewriting to critical sections in particular [12]. A key distinguishing characteristic of our work from these solutions is that we execute directly out of the native binary outside critical sections, thus paying no overhead when there is no instrumentation. We obtain these benefits by exploiting our specific use-case: instrumentation is only necessary inside critical sections. Our instrumentation methodology also bears resemblance to static instrumentation tools [8] since we operate out of a static persistent instrumentation cache. However we sidestep the problem of static instrumentation tools in looking past indirect branches and locating regions to instrument by depending on dynamic execution. We thus implement a hybrid instrumentation scheme.

Bruening et al. [4] implemented a scheme to share and persist code caches across processes, reducing code cache space overheads for common code such as shared libraries. Reddi et al. [15] implemented a scheme to persist the code cache of PIN across multiple runs. This allowed lowering the overhead of using dynamic binary rewriting, particularly for short programs since there was no overhead to re-generate instrumentation. However neither of them consider the possibility of eliminating the dynamic binary rewriting engine altogether as we do for passive instrumentation mode.

Another key aspect of our work is elimination of instrumentation for thread-private locations. We use multiple memory maps to allow access by the owning the thread while preventing accesses by other threads; a similar idea has been explored in work on providing STM with strong atomicity [1]. However, that work required the capability to generate instrumentation for accesses outside transactions at runtime. They also placed the mirror map at a constant offset from the actual page, thus assuming a limit to the size of the heap. We add a level or indirection and in return obtain the capability to dynamically size the heap and unmap mirror mappings that are not needed, thus conserving virtual memory.

8. Conclusion

We have presented a system for instrumenting shared memory accesses that uses a combination of static and dynamic instrumentation, persistence and a custom dispatcher to provide low overhead pay-to-use instrumentation. It has proven particularly efficient for our applications and we believe that it can be extended to more general purpose instrumentation such as instrumenting functions or entire libraries. The low overhead of using a static persistent instrumentation cache would make binary rewriting for program optimisation far more feasible. A persistent instrumentation cache can also be pre-generated and distributed with binaries for useful applications such as on-demand profiling and tracing. We intend to make our infrastructure available to other researchers for use and possible extensions.

We also believe that the techniques in this work can be used to improve existing binary rewriting engines. PIN for example, already provides a mechanism to dynamically attach to and detach from running binaries. A more fine-grained version of this capability where it attaches past a marker function (like a lock acquire) and detaches past another marker function (like a lock release) would bring much of the pay-to-use instrumentation benefits that we have aimed for in this work to PIN.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 185–196, 2009.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, 2000.

[3] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[4] D. Bruening and V. Kiriansky. Process-shared and persistent code caches. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 61–70, 2008.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, 2008.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, 2006.

[7] A. Dragojevic, Y. Ni, and A.-R. Adl-Tabatabai. Optimizing transactions for captured memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 214–222, 2009.

[8] A. Eustace and A. Srivastava. Atom: a flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, pages 303–314, 1995.

[9] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd international conference on Supercomputing*, pages 126–135, 2009.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[11] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, 2007.

[12] M. Olszewski, J. Cutler, and J. G. Steffan. Judostm: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 365–375, 2007.

[13] M. Payer and T. R. Gross. Generating low-overhead dynamic binary translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, pages 22:1–22:14, 2010.

[14] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 173–184, 2009.

[15] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 74–88, 2007.

[16] A. Roy, S. Hand, and T. Harris. Exploring the limits of disjoint access parallelism. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, 2009.

[17] B. Schwarz, S. Debray, G. Andrews, and M. Legendre. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Proceedings of the 2001 Workshop on Binary Translation*, 2001.

[18] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[19] S. Sridhar, J. S. Shapiro, and P. P. Bungale. HDTrans: a low-overhead dynamic translator. *SIGARCH Computer Architecture News*, 35(1):135–140, 2007.

[20] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive locks: Combining transactions and locks for efficient concurrency. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2009.

[21] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the 2005 IEEE International Symposium On Signal Processing And Information Technology*, pages 7–12, 2005.

[22] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 185–196, 2008.

[23] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, 2008.