Retroactive Aspects: Programming in the Past

Robin Salkeld
University of British Columbia
rsalkeld@cs.ubc.ca

Brendan Cully
University of British Columbia
brendan@cs.ubc.ca

Geoffrey Lefebvre
University of British Columbia
geoffrey@cs.ubc.ca

Wenhao Xu
University of British Columbia
zidadi@cs.ubc.ca

Andrew Warfield
University of British Columbia
andy@cs.ubc.ca

Gregor Kiczales
University of British Columbia
gregor@cs.ubc.ca

ABSTRACT

We present a novel approach to the problem of dynamic program analysis: writing analysis code directly into the program source, but evaluating it against a recording of the original program's execution. This approach allows developers to reason about their program in the familiar context of its actual source, and take full advantage of program semantics, data structures, and library functionality for understanding execution. It also gives them the advantage of hindsight, letting them easily analyze unexpected behavior after it has occurred. Our position is that writing offline analysis as *retroactive aspects* provides a unifying approach that developers will find natural and powerful.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering—*Testing and Debugging*

General Terms

Languages, Measurement, Verification

Keywords

Aspect-oriented Programming, Execution Mining, Deterministic Replay, Dynamic Analysis

1. BACKGROUND AND MOTIVATION

A recurring and frustrating problem faced by software developers is bugs that only appear for real customers in released software. Diagnosis techniques such as debuggers and assert statements that are invaluable during development become painful and expensive for deployed systems. Attempts to reproduce the issue often fail because of differences between execution environments or non-determinism.

Execution capture and replay offers the potential to record execution with perfect, instruction-level fidelity and analyze it at some later date [7, 22, 19, 20]. The overhead involved in recording enough data to deterministically replay execution exactly as it originally occurred can be low enough for even commercial virtualization tools to support it [23], making it a real option for deployed systems.

WODA '11, July 18, 2011, Toronto, ON, Canada


```
before(struct mutex * lock) :
        call($ mutex_lock(...)) && args(lock) {
  if (!check_order(current, lock)) {
    printf("Lock order reversal: %p (%s)", lock, lock->name);
  }
}
before(struct mutex * lock) :
        call($ mutex_unlock(...)) && args(lock) {
  remove_from_order(current, lock);
}
```

Figure 1: A simple example. The `check_order` and `remove_from_order` functions manipulate a lock order relationship tree; their implementation is omitted for brevity.

However, while these technologies provide the primitives for post hoc execution analysis, they do not yet address the problem of how average developers can effectively author and evaluate such analysis. Execution traces are far too low-level to be of any use to developers whose expertise is in a high-level language. Omniscient debugging [15, 21] helps to bridge the semantic gap by providing a debugger-like trace browser that can move backwards as well as forwards through program execution, but its power is limited since it cannot evaluate complex expressions nor maintain state between breakpoints. To diagnose complex bugs or collect application-specific statistics, often the simplest approach is just to instrument the program with more code, but unfortunately deterministic replay processes are easily broken by attempts to instrument them.

Ideally, offline analysis should be just as intuitive and familiar as adding `printf()` statements within a program's source code: the analysis' interaction with the original program should be type-safe; consistent with its value, name binding and control flow semantics; and able to reuse its datatypes and algorithms. In addition, it should be possible to author analysis code with consistent semantics that do not depend on which execution capture and replay technology is used, or even on whether the analysis is evaluated during the program execution or after it.

Our position is that providing these kinds of restrictions and guarantees are natural programming language design problems, and should be solved by extending the original programming language. In particular, we propose to de-

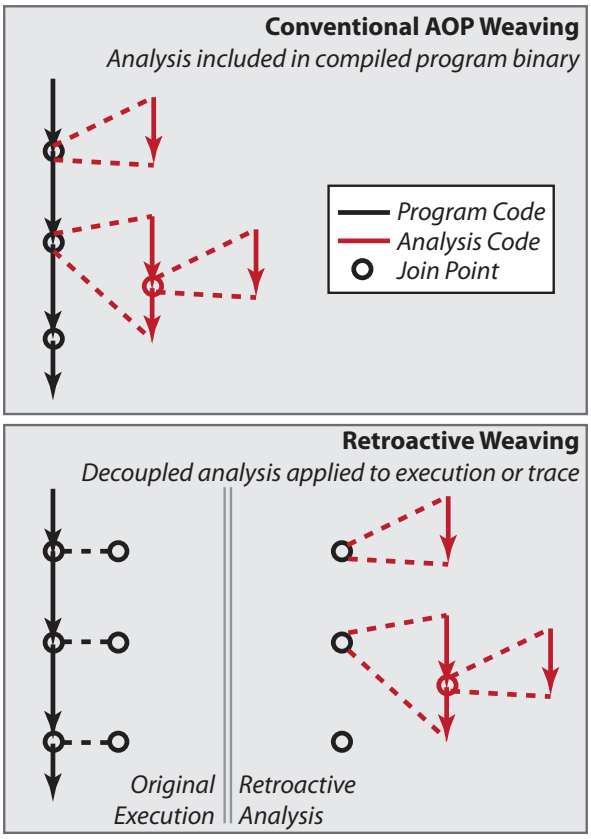

Figure 2: Traditional advice weaving versus retroactive advice weaving. We propose to restrict analysis advice such that its semantics are indistinguishable between the two implementations.

fine a more restrictive semantics for the aspect-oriented programming concepts of pointcuts and advice such that their online and offline behaviour is indistinguishable. We refer to such aspects as *retroactive aspects*, and to evaluating such aspects in relation to a previous execution after the fact as *retroactive weaving*, in contrast to static or dynamic weaving as illustrated in Figure 2. We show that verifying and retroactively weaving such aspects can be accomplished through static analysis and code transformation techniques, and claim that this extension allows AOP to provide a unified dynamic analysis framework that applies equally well in the present or the past.

2. RETROACTIVE DYNAMIC ANALYSIS

This section discusses the challenges that arise in trying to provide a friendly post hoc dynamic analysis system, and how our definition of retroactive aspects addresses them. For the purposes of concrete discussion we will focus on applying the approach to the aspect-oriented C variant AspeCtC (ACC) [2], since our initial prototype was developed using this language in order to analyze the Linux kernel. Figure 1 contains a sample ACC aspect which we will use as a running example. It is inspired by the `witness` kernel module, which validates that lock acquisition and release code is structured correctly to avoid deadlocks by verifying that lock ordering is consistent. This is one of several units of conditionally-enabled kernel code that provides valuable debugging information but is normally too slow or intrusive to enable for production builds.

2.1 Compatibility with Program Source

Our primary goal is to allow analysis code that is as close as possible to the program's source code. Post hoc analysis code needs to specify individual points in time of interest in the program's execution to trigger additional code, and have access to local values that aren't available from the global state. Offering the same flexibility as inserting additional code at arbitrary locations in the source code is challenging, and ensuring the instrumentation is run exactly when needed often involves complicated and flow-sensitive specifications.

This problem is addressed in various ad-hoc ways by existing dynamic analysis tools, both online or offline. PIN [16] and the ATOM language it is based on [24] are quite aspect-like at the instruction level, but each instruction must be individually inspected as a metadata object to determine where to insert additional instructions. The Program Trace Query Language (PTQL) [12] is a reduced dialect of SQL customized to query program executions, which is described as being equally applicable to post hoc evaluation. PTQL treats individual execution events such as function calls as relational tuples, with time as an explicit dimension that must be manually manipulated, which makes analysis of control flow awkward to specify. VMWare's VAssert [1] system supports replay-only statements in source code, which requires analysis foresight and is less convenient for testing bug theories after the fact.

Pointcuts provide a rich specification language for solving exactly this problem, and are equally applicable to matching joinpoints that occurred in the past. They include binding mechanisms such as `args(x, y)` and `return(z)` which allow values to be extracted from the original program and used in analysis code. Using a declarative sub-language to describe analysis trigger points also presents opportunities for optimization in the context of execution tracing and replay, since irrelevant events and even entire periods of execution can be ignored or skipped over when reading a trace or instrumenting a dynamic replay process.

Providing retroactive evaluation semantics for an existing AOP language also enables post hoc evaluation of higher-level runtime analysis tools that are built on top of that language. If post hoc evaluation of AspectJ is possible, for example, then tools such as the monitor-oriented programming framework JavaMOP [5] that are implemented by producing AspectJ code can also be run post hoc. This is only true if the aspects such tools produce are guaranteed to be valid retroactive aspects (as described below); verifying this is a potential avenue of future work.

In some cases an AOP language's pointcut types may not express all of the desired concepts for post hoc analysis, but as proposed previously [9, 4] it is reasonable to extend pointcut languages for greater expressiveness or precision; we have extended ACC ourselves, in fact, in order to address its shortcomings when applied to kernel-level code. Implementing AOP concepts beyond pointcuts and advice in the context of retroactive evaluation could also enable more sophisticated analysis. Inter-type declarations in particular offer an attractive solution for attaching additional analysis

state to program structures after the fact, and are flagged as future work in our prototype.

2.2 Access to Program State

Retroactive advice will read values from the original program execution wherever such values are referenced through pointcut arguments or global variables. Retroactive weaving therefore needs to extract those values from the original execution.

Simply applying traditional dynamic advice weaving to a deterministic replay process would appear to be an ideal implementation, but unfortunately this will inevitably interfere with the fragile replay process. In our case of kernel-level process replay, for example, it is necessary to record all system calls made to the kernel so they they can be reproduced in replay. Any system calls made by the analysis code will be incorrectly trapped by the replay process and produce the wrong results, and in addition throw off future results, so even an apparently harmless call to `printf()` can be disastrous. Similar issues exist for any replay mechanism - in general efficient replay relies on being able to assume all deterministic behaviour remains exactly the same between record and replay.

Post hoc analysis must therefore be evaluated in a separate process. Moreover, because the analysis is compatible with the program source and can maintain its own state, references to both the original program's state and the analysis state must be disambiguated. In C, this means that pointer values may actually refer to either the original program's memory or the new analysis runtime's memory. We refer to this as the *dual address space problem*.

In other decoupled analysis frameworks such as Speck [19], data must be manually marshalled between processes in order to synchronize them, which does not scale well to arbitrary source-level analysis. Choi and Alpern use *remote reflection* [18] to debug a Jalapeño JVM [6] from another JVM. Remote reflection solves the dual address space problem in Java dynamically by customizing the debugging JVM to track whether each object is remote or local; references to remote object members are handled by communicating with the JVM being debugged, and the flag is propagated through such references. This approach requires a custom analysis runtime or virtual machine, and in the case of C the overhead of maintaining fat pointers to attach these flags would have been prohibitive.

Our approach is to solve the problem statically instead. At compile time, variables declared by the program source or bound by pointcuts are marked by our aspect compiler as program state. Other variables and fields are inferred based on dataflow as either program or analysis state within the source code, and references to program state are rewritten to instead call a runtime API to recreate that state. In Figure 1, for example, the `lock` pointer is bound though the `args` pointcut, and is hence marked as referring to base state, as is `lock->name`. So is the result of `current`, a Linux kernel macro used to get a pointer to the currently running task from a CPU local variable. Our implementation of this inference process is described in Section 3.1.

2.3 Avoidance of Side-Effects

For source-level post hoc analysis to have consistent semantics whether it is evaluated online or offline, the analysis code must not have side-effects that would have changed the program behaviour; the analysis framework should reject any such analysis.

```
// Program code
void fetchFoo(char * foo) {
    increaseFooRefCount();
    computeFoo(foo);
}

// Analysis advice
after() : call(bar()) {
    char foo[100];
    fetchFoo(foo);
    printf("Foo is: %s", foo);
}
around() : cflow(adviceexecution())
           && call(increaseFooRefCount()) {}
```

Figure 3: An example of supressing unwanted side-effects using around advice.

In other systems where analysis is performed outside the original program, such as Speck or remote debugging, any side-effects occur externally as well and are not in danger of perturbing the replay. However, allowing them can cause later analysis code to observe these changes and behave inconsistently, and these deviations can be buried deep within application code called from the analysis and hence very subtle. Other approaches based on virtualization such as Introvirt [13] and VAssert [1] allow mutations to occur within the replay process, but then revert to a prior checkpoint to undo their effects before continuing. This also prevents the analysis from maintaining any state between triggers, however, and hence restricts its power.

The same static analysis that addresses the dual address space issue can also be used to detect and reject attempts to write to the program's memory space, which is the most common way analysis code could affect the program's execution. A large percentage of useful application code, however, will include side-effects such as caching even if their primary use is to read state, which would seem to imply they cannot be used in retroactive aspects.

One solution is to use a generic pointcut introduced in AspectJ called `adviceexecution()`, which matches all joinpoints inside advice code. Combining it with the `cflow` pointcut operator creates a pointcut that covers all advice execution. This is often used to exclude aspects from applying to other aspects when the interaction is undesirable. In the case of retroactive aspects, it therefore covers all code in the decoupled analysis, and so can be used to only suppress or redirect side effects in application code when called retroactively. This achieves the goal of advice code that behaves identically whether woven directly or retroactively. See Figure 3 for an example. Here around advice is used to replacing undesired side effects by replacing certain calls with no-ops.

We are continuing to investigate the best approach for managing side-effects. It is unclear how to rigorously permit intentional, safe side-effects in retroactive advice: outputting results to the console or the file system should be permitted, for example, even though such behaviour technically could have been observed by the program. In addition, it may be too onerous to omit all side-effects in reused program code in some cases, and we may explore structured fork-and-revert semantics similar to VAssert for advice. Defining clean semantics that allow values to escape

the rollback process in C is quite challenging, however, since it is unclear how to deal with pointers.

3. IMPLEMENTATION

This section defines our retroactive weaving prototype for ACC. The system consists of a compiler that compiles ACC aspect files into a C library, and two different runtimes for evaluating the compiled aspects against a particular program execution. In combination with the existing ACC weaver, this allows the same ACC code to be evaluated either inline or post hoc.

3.1 Compiler

Our compilation process consists of a chain of several distinct source transformation phases taking ACC code as input and producing C code as output, followed by a call to an underlying C compiler (gcc in our case) to produce object files. The source transformation phases are implemented by extending the C Intermediate Language OCaml library [17], which is designed to support the analysis and transformation of C source.

We have modified the CIL distribution to support parsing ACC code and representing pointcuts and advice in its abstract syntax tree. The compiler also annotates variables and types to track which memory space they refer to by leveraging CIL's support for gcc attributes, which are declarations with the syntax "`$attribute`" that can be attached to nearly all elements of C syntax. In our case these annotations are either resolved `$base` or `$aspect` annotations, or annotation variables such as `$$a` that represent unknowns. Since the C type system includes value types with multiple layers of indirection (i.e. pointers to pointers), the annotations are attached to all levels of types; As an example, the type `char $base * $base * $aspect` is interpreted to mean "pointer in the aspect space to a location storing a pointer in the base program space containing an immutable character derived from program values." This would be exactly the correct type to ascribe to a variable holding the beginning of a sequence of strings extracted from the original execution: the array itself would be a consecutive region of space in the aspect execution, but the pointer values inside would refer to the memory of the original execution.

The source transformation phases are outlined below.

3.1.1 Annotation Inference

This phase walks the ACC source tree in a bottom-up pattern, inserting annotations and variables as needed. The `$base` and `$aspect` annotations are first introduced as base cases into the AST according to the semantics of weaving: values that are bound from the original execution through pointcut parameters (e.g. `args(a, b, c)` and `this->args`) are assigned to the `$base` space at all levels of indirection, whereas values that are allocated by the weaving runtime (i.e. `targetName(x)` and `this->targetName` are similarly assigned to the `$aspect` space.

This phase also builds up a list of type constraints of the form `T1 = T2` based on nodes in the AST that require two address spaces to be the same: assignments, function arguments and return values, arithmetic, etc. In general values from different address spaces cannot be used together, although an important exception is pointer arithmetic wherein offset values can be from any address space, and the base pointer determines the address space of the result. Annotation variables are then resolved by solving the constraints using the W unification algorithm [10].

3.1.2 Annotation Propagation

As an additional aid to determining the address spaces of values in the source, this phase takes advantage of the observation that addresses in the original program address space cannot point to values derived from the aspect space by construction; to arrive in such a state requires an assignment to a target lvalue, which is prohibited as an illegal side-effect. Therefore, the occurrence of any type variable as an address space annotation in a type at a deeper level of indirection than a `$base` annotation can be replaced by `$base` as well. For example, `char $$a * $base * $aspect` will become `char $base * $base * $aspect`.

This is an important rule for C code, and kernel code in particular, in which it is common to calculate addresses through raw numeric calculations followed by a cast to a pointer. Ideally this rule could be incorporated into the general inferencing phase, but this rule cannot be encoded as a simple equality of types and hence cannot be included in a sound way. As future work we plan to use a less simplistic inferencing algorithm in order to address this lack of power.

3.1.3 Program State Access

If the previous phases have not rejected the input program, the AST now contains only language constructs whose interactions with the program state are fully tractable. The next phase then replaces all instructions that read the program's memory with calls to reconstruct the values from the retroactive weaving runtime. This includes the reads of any lvalue derived from the program execution, or taking the address of any such lvalue. Addresses of symbols and data structure offsets are calculated from information extracted from a debug build of the program. The functions used to produce addresses of global variables and resolve register and memory accesses are declared in a header file added to the source at this stage, to be implemented by the particular backend weaving runtime.

Some special cases are implemented directly in the compiler backend since C is not expressive enough to redefine code using around advice as described earlier. For example, `printf` takes a variable number of arguments, and the operations it performs on those arguments depend on the format string; a pointer value matched with a `%s` pattern will be dereferenced, but one matched with a `%p` will not. Our solution is to match the formats to the arguments and to copy values like strings into aspect space if `printf` will dereference them, and then pass the modified arguments to the original implementation. This supports calls that pass a mix of program and analysis pointers, which is useful given that some joinpoint data consists of analysis pointers (e.g. function names as `char *` values).

3.1.4 Transformation from ACC to C

The final component is responsible for splitting the aspect bodies from their associated pointcuts. For a single given retroactive aspect, this component produces a C source file with three separate artifacts:

1. A set of advice bodies transformed into regular functions by discarding their pointcuts and advice kind, along with stub methods for invoking the advice body functions with a unified interface;

2. A function table for these stub methods; and

3. A string constant containing the set of aspects from the input in ACC syntax, with their bodies discarded, in the same order as in the function table.

This enables parsing the aspect stubs and dynamically invoking their body functions without recompilation of the target weaving runtime.

3.2 Runtime

This section describes the interface between the aspect runtime and the target execution environment and a brief description of the two target environments we currently support: one instantiates state on demand from an instruction level trace produced by the Tralfamadore [14] dynamic analysis framework, and one provides hooks into the virtual machine monitor QEMU [3], which we have modified to perform deterministic recording and extract virtual machine state during replay. For these backends, the retroactive aspect compiler produces a shared library containing the aspects in the original ACC source as regular C functions, with metadata consisting of the advice kind and pointcuts attached. This shared object runtime exports a common retroactive weaving interface, which serves two purposes: first, to provide the target environment with event notification callbacks on events such as function calls and returns and context switches to produce join points, and second, to let it register functions for inspecting target register and memory state.

3.2.1 Understanding Kernel Execution

Running aspects against kernel execution is made somewhat more complicated by context-sensitive pointcuts such as `cflow(pc)`, which matches any join point that occurs inside, or beneath, the pointcut `pc`. User-level aspects can refer to the threading abstraction provided by the operating system to uniquely identify individual flows of execution (i.e. `pthread_self()`) and track joinpoint stacks correctly.

Kernel code is more challenging because although the notion of thread exists, it is not a good abstraction for tracking individual flows of execution due to interrupts and exceptions. Interrupt handlers execute in the context of the currently running thread but are conceptually different flows of execution. Identifiers such as `current`, which maps to the currently executing task, cannot be used to distinguish between threaded (system call, kernel threads) and interrupt handler execution. Interrupts can also be nested making this problem worse.

Because our trace-based framework is designed to analyze kernel execution, it provides an abstraction to demultiplex individual flows of kernel execution. It uses platform-specific rules to isolate system calls, interrupts, exceptions and kernel threads and label them with a unique opaque identifier. These rules are both hardware and operating system specific and require tracking stack switching, interrupts, exceptions and instruction such as `iret` and `sysexit`.

We have extended the implicit structure encoding the current join point to support the expression `this->cflowID`. This evaluates to the flow identifier provided by the backend runtime that can be used to index per-control-flow information for later retrieval, while still encapsulating the details of how independent control flows are tracked. We believe this to be a logical, generic extension to AOP and suggest that it could be used in any pointcut and advice language to avoid dependencies on specific threading libraries.

3.2.2 Trace-based Runtime

The Tralfamadore backend is completely offline in that running aspects involves no re-execution of guest code at all. All updates to register and memory that occurred during execution recording are present in the trace. Because Tralfamadore uses a modified version of QEMU to capture traces, it can be used to record the execution of an unmodified operating system kernel.

To add support for retroactive advice execution, we implemented a new top level operator stacked on top of the existing kernel flow and function call operators. This new operator implements a driver loop that pulls on these events and calls into the weaving runtime whenever an event of interest is recognized. Tralfamadore also supports tracking the state of registers and provides a memory index which supports efficiently finding the last update to a given memory address range. We use both of these features to support callbacks from the weaving runtime to inspect guest state.

3.2.3 Deterministic Replay Runtime

We have also implemented a retroactive weaving runtime directly into a virtual machine monitor (QEMU) which we have enhanced to perform deterministic execution recording and replay [11]. It records all non-deterministic events (such as external interrupts or reads of the CPU timestamp counter) occurring during execution so that they may be injected at the same point in execution during replay.

We have also added hooks into QEMU to register callbacks that can be invoked before and after the execution of any basic block. We use these hooks to track individual flows of kernel execution similarly to Tralfamadore and to track function calls and return. Whenever such an event occurs, these hooks call into the retroactive weaving library, potentially calling aspect code. Accessing guest state is much more efficient than in the trace-based approach: it is simply a matter of mapping pointers into QEMU's data structure representing the memory of the target virtual machine. Compiled aspects and the retroactive weaving runtime are loaded and unloaded into QEMU during replay using the standard dynamic loaded library mechanism.

Deterministic recording and replay can be made very efficient through the use of CPU performance counters [11], costing as little as 5% overhead in VMWare [23]. Our prototype is much more expensive (approximately 20x) due to its pure software implementation, which makes it easier to hook instrumentation into replayed execution. As AfterSight [8] demonstrated, it is possible to use low-overhead hardware deterministic recording as the source for software replay, and we hope to do this ourselves in future work.

4. FUTURE WORK

Because our tracing framework is limited to kernel memory space, our weaving tool is also limited to advising kernel source code. The same techniques apply equally well to user-level code, however, and we intend to extend our stack to cover application source code as well as OS code. This will open up many avenues for complex analysis, especially in the areas of gathering statistics and searching for invariant violations related to processes and threads.

We are in the process of porting several dynamic analysis tools for Linux into aspects so that they can be applied retroactively to deployed systems. We also plan to apply the same approach to AspectJ using a suitable replay mechanism for Java, in order to demonstrate the generality of retroactive AOP and to gain experience with applying it to a more high-level language.

We intend to explore the possibility of also using pointcuts and advice to explore hypothetical execution possibilities by explicitly switching from deterministic replay to live execution at a well-defined point. This would enable testing potential bug fixes and patches while still holding the execution up to the fork point fixed.

5. CONCLUSIONS

We have presented the concept of retroactive advice as a unified, source-level approach to post hoc dynamic analysis. We have also described our prototype implementation of a retroactive advice weaver for ACC and identified areas for future growth. We assert that running advice on prior executions of a program opens up a wide range of analysis applications that might otherwise be infeasible.

6. ACKNOWLEDGEMENTS

This paper is based upon work supported in part by an NSERC CGS M fellowship.

7. REFERENCES

[1] VAssert programming guide, 2008.

[2] ACC: the AspeCt-oriented c compiler. http://research.msrg.utoronto.ca/ACC, 2009.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, 2005.

[4] E. Bodden and K. Havelund. Racer: effective race detection using aspectj. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 155–166, New York, NY, USA, 2008. ACM. ACM ID: 1390650.

[5] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *ACM SIGPLAN Notices*, OOPSLA '07, pages 569–588, New York, NY, USA, 2007. ACM. ACM ID: 1297069.

[6] J. Choi and B. Alpern. DejaVu: Deterministic Java Replay Debugger for Jalapeno Java Virtual Machine. *OOPSLA 2000 Companion*, 2000.

[7] J. D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, 2001.

[8] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, 2008.

[9] J. Cook and A. Nusayr. Using AOP for Detailed Runtime Monitoring Instrumentation. In *WODA 2008: the sixth international workshop on dynamic analysis*.

[10] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages*, Albuquerque, New Mexico, 1982.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *Operating Systems Design and Implementation*, 2002.

[12] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Object-Oriented Programming, Systems, Languages, and Applications*, page 402, 2005.

[13] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 91–104, New York, NY, USA, 2005. ACM.

[14] G. Lefebvre, B. Cully, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Tralfamadore: unifying source code and execution experience. In *EuroSys*, 2009.

[15] B. Lewis. Debugging backwards in time. In *Automated and Analysis-Driven Debugging*, 2003.

[16] 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 *Programming Language Design and Implementation*, 2005.

[17] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*. 2002.

[18] T. Ngo and J. Barton. Debugging by remote reflection. In *Euro-Par 2000 Parallel Processing*, pages 1031–1038, 2000.

[19] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Architectural Support for Programming Languages and Operating Systems*, 2008.

[20] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7, New York, NY, USA, May 2005. ACM. ACM ID: 1083251.

[21] G. Pothier, É. Tanter, and J. Piquer. Scalable omniscient debugging. *ACM SIGPLAN Notices*, 42(10), 2007.

[22] V. Schuppan, M. Baur, and A. Biere. JVM independent replay in java. *Electronic Notes in Theoretical Computer Science*, 113:85–104, Jan. 2005.

[23] M. X. Sheldon, G. V. Weissman, and V. M. Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Modeling, Benchmarking and Simulation*, 2007.

[24] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, Orlando, Florida, United States, 1994. ACM.