Javana: A System for Building Customized Java Program Analysis Tools

Jonas Maebe Dries Buytaert Lieven Eeckhout Koen De Bosschere
ELIS, Ghent University, Belgium
{jmaebe,dbuytaer,leeckhou,kdb}@elis.UGent.be

Abstract

Understanding the behavior of applications running on high-level language virtual machines, as is the case in Java, is non-trivial because of the tight entanglement at the lowest execution level between the application and the virtual machine. This paper proposes Javana, a system for building Java program analysis tools. Javana provides an easy-to-use instrumentation infrastructure that allows for building customized profiling tools very quickly.

Javana runs a dynamic binary instrumentation tool underneath the virtual machine. The virtual machine communicates with the instrumentation layer through an event handling mechanism for building a vertical map that links low-level native instruction pointers and memory addresses to high-level language concepts such as objects, methods, threads, lines of code, *etc*. The dynamic binary instrumentation tool then intercepts all memory accesses and instructions executed and provides the Javana end user with high-level language information for all memory accesses and natively executed instructions.

We demonstrate the power of Javana through a number of applications: memory address tracing, vertical cache simulation and object lifetime computation. For each of these applications, the instrumentation specification requires only a small number of lines of code. Developing similarly powerful profiling tools within a virtual machine (as done in current practice) is both time-consuming and error-prone; in addition, the accuracy of the obtained profiling results might be questionable as we show in this paper.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Tracing; D.3.4 [*Processors*]: Run-time Environments

General Terms Experimentation, Measurement, Performance

Keywords Customized Program Analysis Tool, Java, Aspect-Oriented Instrumentation

1. Introduction

Understanding the behavior of software is of primary importance to improve its performance. Application and system software developers need a good understanding of an application's behavior in order to optimize overall system performance. Analyzing the behavior of applications written in languages such as C and C++ is a well understood problem. However, understanding the behavior of modern software that relies on a runtime system, also called a virtual machine (VM), is much more challenging. The popularity of high-level language virtualization software has grown significantly over the recent years with programming environments such as Java and .NET. The reasons for the increased popularity of high-level language virtual machines are portability, security, robustness, automatic memory management, *etc*. Virtualization though makes the behavior of modern software fairly hard to understand because of the tight entanglement between the application and the virtualization software.

1.1 The Javana concept

This paper proposes Javana, a system for building customized Java program analysis tools. Javana comes with an easy-to-use instrumentation framework so that only a few lines of instrumentation code need to be programmed for building powerful profiling tools. The Javana instrumentation framework provides the end user with both high-level and low-level information. The high-level information relates to the Java application and the VM, such as thread IDs, method IDs, source code line numbers, object IDs, object types, *etc*. The low-level information consists of instruction pointers and memory addresses. Running the Java application of interest within the Javana system along with user-specified instrumentation routines then collects the desired profiles of the Java application.

The Javana system consists of a VM along with a dynamic binary instrumentation tool that runs underneath the VM. The virtual machine communicates with the dynamic binary instrumentation tool through an *event handling* mechanism. The virtual machine informs the instrumentation layer about a number of events, for example when an object is created, moved or collected, or when a method gets compiled or re-compiled, *etc*. The dynamic binary instrumentation tool then catches these events and subsequently builds a *vertical map* that links instruction pointer and memory addresses to high-level language concepts.

The dynamic binary instrumentation tool also captures all natively executed machine instructions during a profiling run within Javana; this includes instructions executed in native functions called through the Java Native Interface (JNI). Instrumenting all natively executed machine instructions causes a substantial slowdown, however, it enables Javana to know for all native instructions from what method and thread the instruction comes and to what line of source code the instruction corresponds; and for all accessed memory locations, Javana knows what objects are accessed.

The Javana concept is also easy to transfer to other virtual machines and other dynamic binary instrumentation tools; building a Javana system is easy to do. Only a few lines of code need to be added to the virtual machine to make the virtual machine Javana-enabled as we demonstrate through our proof-of-concept implementation that combines the Jikes RVM [1] with DIOTA [16, 17].

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-348-4/06/0010. . . $5.00.

Our proof-of-concept Javana implementation is publicly available at `http://www.elis.ugent.be/javana/`.

1.2 Applications

Javana enables the building of vertical profiling tools, *i.e.*, profiling tools that crosscut the Java application, the VM and the native execution layers. Vertical profiling tools are invaluable for gaining insight into the overall performance and behavior of a Java application. When looking at the lowest level of the execution stack, *i.e.*, when looking at the individual instructions executed on the host machine, it is hard to understand the application's behavior because of the fact that the virtualization software gets intermixed with application code.

However, when the goal is deep understanding of the application's behavior, the lowest level of the execution stack really is the level to look at. Vertical profiling enables gaining such insights and Javana makes vertical profiling easy to do. Building equally powerful profiling tools without Javana is both tedious and error-prone; modifying the virtual machine by adding instrumentation code changes the code and data layout which perturbs the native execution behavior substantially. Dynamic binary instrumentation underneath the virtual machine as done in Javana alleviates this issue.

In this paper we demonstrate the power of Javana through three applications. Our first application is memory address tracing. A recent study published by Shuf *et al.* [21] analyzed the memory behavior of Java applications based on memory address traces. They instrumented the virtual machine to trace all heap accesses, but did not trace stack accesses. We found that on average 58% of all memory accesses in a Java application are non-heap accesses. As such, not including non-heap accesses in a memory behavior analysis study may significantly skew the overall results. The Javana system captures all memory accesses and consequently is more accurate.

In our second application we build a vertical profiling tool for analyzing the memory hierarchy behavior of Java applications. This cache performance profiling tool tracks cache miss rates per object type and per method and thus allows for quickly computing the top most cache miss causing lines of code, the top most cache miss causing object types, *etc*. This is invaluable information for an application developer who wants to optimize the memory performance of his software. Again, we want to emphasize how easy this profiling tool was to set up — only a few lines of instrumentation code are needed.

Our third application shows how easy it is to build an object lifetime analysis tool in Javana. Previous work [20] has shown that object lifetime is an important characteristic which can be used for analyzing and optimizing the memory behavior of Java applications. Computing an object's lifetime, although conceptually simple, is challenging in practice without Javana because the virtual machine needs to be adjusted in numerous ways in order to track all possible accesses to all objects, including accesses that occur through the Java Native Interface (JNI). This requires an in-depth understanding of the virtual machine. Computing object lifetime distributions with Javana on the other hand, is easy to set up and in addition, is guaranteed to deliver accurate object lifetimes.

1.3 Paper organization

This paper is organized as follows. We first detail on the Javana system and subsequently describe the Javana instrumentation language that we developed as part of the Javana system. We then quantify Javana's performance and demonstrate how powerful Javana is for quickly building customized Java program analysis tools. Finally, we discuss related work and conclude the paper.

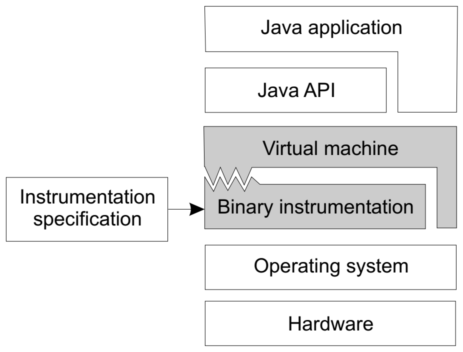

Figure 1. The Javana system for profiling Java applications.

2. The Javana system

Figure 1 illustrates the basic concept of the Javana system. The top of the execution stack shows a Java application that is to be profiled. The Java application together with a number of Java libraries runs on top of a virtual machine. The virtual machine translates Java bytecode instructions into native instructions. A dynamic binary instrumentation tool resides beneath the virtual machine and tracks all native instructions executed by the virtual machine.

The key point of the Javana system is that the virtual machine informs the dynamic binary instrumentation tool through an event handling mechanism whenever an object is created, moved, or deleted; or a method is compiled, or re-compiled; or a thread is created, switched or terminated. The dynamic binary instrumentation tool then uses these events to build *vertical maps* that associate native instruction pointers and memory addresses with objects, methods, threads, *etc.* The dynamic binary instrumentation tool also intercepts all memory accesses during the execution of the Java application on the virtual machine. This includes instructions executed in native JNI functions, but excludes kernel-level system calls as will be discussed later. Using the vertical maps, the binary instrumentation tool associates native machine addresses to high-level concepts such as objects, methods, *etc*. This high-level information along with the low-level information is then made available to the end user through the Javana instrumentation framework.

The remainder of this section discusses the Javana system in more detail. We discuss the events that are triggered by the virtual machine, the dynamic binary instrumentation layer in the Javana system, the event handling mechanism, the vertical instrumentation, the perturbation of the Javana system and finally our Javana proof-of-concept implementation. All of the subsections below give more-or-less a general description of what the issues are for building a Javana system; the final subsection then discusses our own proof-of-concept implementation.

2.1 Events triggered by the virtual machine

The Javana system requires that the virtual machine is instrumented to trigger events. These events communicate information between the virtual machine and the dynamic binary instrumentation tool. Our current Javana system supports the following events:

- Class loading: When a new class is loaded and a new object type becomes available, the new class name is communicated to the binary instrumentation tool.
- Object allocation: When a new object is allocated, the object's type and memory location (object starting address and its size) are communicated.

- Object relocation: When an object is copied by the garbage collector, the object's new location is communicated to the instrumentation tool.
- Method compilation: When a method is compiled, its name, memory location and a 'code to line number' map are communicated to the instrumentation tool.
- Method recompilation: When a method is recompiled, the method's location and 'code to line number' map are updated in the binary instrumentation tool.
- Method relocation: When code is moved by the garbage collector, the code's new location in memory is communicated.
- Memory freed during garbage collection: When memory is freed, the address range of the freed memory space is communicated to the binary instrumentation tool.
- Java thread creation: When a new Java thread is created, the thread's ID and name are communicated.
- Java thread switch: When a Java thread switch occurs, the newly scheduled Java thread's ID is communicated.
- Java thread termination: When a Java thread has ended execution, this is communicated to the dynamic binary instrumentation tool.
- Java thread stack switch: When a Java thread stack is relocated, the thread ID, the old stack location and the new stack location are communicated.

Note that this event list is just an example event list that could be tracked within a Javana system. Additional events could be defined and added to this list if desired; implementing events in a virtual machine is easy to do. We found though that this list of events is sufficient for our purpose of building powerful Java program analysis tools, as will be shown in the remainder of this paper.

2.2 Dynamic binary instrumentation

A dynamic binary instrumentation tool takes as input a binary and an instrumentation specification. The binary is the program of interest; this is the Java application running on a virtual machine in our case. The instrumentation specification indicates what needs to be instrumented in the binary; this is going to drive the customized profiling. The dynamic binary instrumentation tool then instruments the program of interest at run time. Upon the first execution of a given code fragment, the instrumentation tool reads the original code, modifies it according to the instrumentation specification and stores the result as part of the instrumented binary. The instrumented version of the code is then executed and the desired profiling information is collected while executing the instrumented binary.

The data memory addresses referenced by the loads and stores in the instrumented binary are identical to the uninstrumented binary. By keeping the original binary in memory at its original address while generating the instrumented binary elsewhere, the instrumented binary obtains correct data values from the original uninstrumented binary in case data-in-code is read. The instrumentation tool also keeps track of correspondences between instruction pointers in the original binary versus the instrumented binary. By doing so, the instrumentation routines see instruction pointers and memory addresses as if they were generated during the execution of the original binary.

Running a dynamic binary instrumentation tool underneath a virtual machine requires that the instrumentation tool can deal with self-modifying code. The reason is that most virtual machines implement a dynamic optimizer that detects and (re-)optimizes frequently executed code fragments. A similar issue occurs when garbage is collected; copying collectors may copy code from one memory location to another. This requires that the dynamic instrumentation tool invalidates the old code fragment and replaces it with an instrumented version of the newly generated code fragment.

Previous work [16] has proposed various approaches to instrumenting self-modifying code. These approaches vary in granularity for tracking self-modifying code: some approaches track memory page level accesses, other approaches track individual memory operations. The bottom line is that all of them cause a substantial slowdown in execution time of the instrumented binary, in some cases to up to a factor 20. However, since the virtual machine communicates with the dynamic binary instrumentation tool through an event handling mechanism, we can optimize the self-modifying code support. We use the information provided by the event handling to invalidate an old code fragment and to replace it with an instrumented version of the new code fragment. This eliminates the slowdown for supporting self-modifying code almost completely.

Note that the dynamic binary instrumentation tool does not track kernel-level system calls. This limits the use of Javana to user-space instrumentation.

2.3 Event handling

The virtual machine triggers events by calling empty functions; these empty functions are native C functions. The dynamic binary instrumentation tool intercepts such function calls and in response calls the appropriate event handlers. Event handlers can accept arguments because the arguments placed on the stack by the virtual machine are available to the binary instrumentation tool as well. For example, when allocating an object, the virtual machine calls the `AllocateObject` function with a number of arguments, namely the object type `t`, its address `m` and its size `s`. The dynamic binary instrumentation tool intercepts such events by inspecting the target addresses of the function calls. If the target address corresponds to the `AllocateObject` function in the above example — the dynamic binary instrumentation tool knows this function by name from the symbol information of the virtual machine — the dynamic binary instrumentation tool transfers control to the appropriate event handler which in turn reads the arguments from the stack and adds this information to its internal data structures. When the event handler has finished execution, control is transfered to the return address of the event's function call, *i.e.*, the instrumented binary gets control again.

Event handling enables the dynamic instrumentation tool to build the *vertical map*. In the above example with the `Allocate-Object` event, the event handler adds the following information to the vertical map: an object of type `t` is allocated in the memory address range `m` to `m+s`. Similar event handlers exist for all the events mentioned in section 2.1.

2.4 Vertical instrumentation

The dynamic instrumentation tool captures *all* native instructions and memory accesses from both the application and the virtual machine during the execution of a Java application within Javana. The vertical map then enables the dynamic binary instrumentation tool to know for each memory access what object is being accessed and what the object's type is; and for every instruction pointer, the dynamic binary instrumentation tool knows to what method, to what line of source code and to what thread the instruction corresponds. The end result is that Javana allows for easily tracking *all* Java object accesses, which is much harder to do without a vertical map and dynamic binary instrumentation support.

In our proof-of-concept Javana system we keep track of the vertical map using two AVL trees — an AVL tree is a self-balancing binary search tree. The first AVL tree, the *method tree*, contains mapping information between instruction pointers and method information. A node in the method tree is identified by an instruction pointer address range that corresponds to a line of source code. Instruction pointer ranges that are not represented in the method tree do not correspond to a Java source code line. The second AVL tree called the *object tree* contains object information. A node in the object tree identifies an object based on the object's address range, *i.e.*, the object's address and size. The remaining address ranges refer to non-objects.

Note that the method and object trees are accessed very frequently during a profiling run. For example, for every memory access the object tree needs to be searched for the corresponding object. This is very time-consuming and has a big impact on the overall profiling overhead. We therefore optimized the accessing of the method and object trees by adding a caching mechanism that tracks recently accessed object and method information. We obtain an average hit rate for the object and method tree caches of 67% and 99%, respectively. In addition, we further optimize the miss case by searching the tree starting from the previous hit. This reduces the tree search time thanks to spatial locality.

2.5 Perturbation

An important property of any instrumentation framework is that the results that are obtained during profiling may not suffer from perturbation. The end user wants the instrumentation framework to be completely transparent to its user, *i.e.*, the instrumentation framework should not impact the results from profiling.

More in particular, in our Javana system, care needs to be taken so that the profiling results are not perturbed by the event handling mechanism. Recall that the virtual machine triggers events by calling an empty method using a number of arguments. Computing the arguments, pushing them onto the stack, and finally calling the empty method introduces some overhead. Since the dynamic binary instrumentation tool instruments all natively executed instructions, the instructions executed for triggering an event in the virtual machine get instrumented as well. In order to alleviate this issue, and to remove any perturbation because of the event handling mechanism, we communicate the address ranges of the virtual machine code for event triggering. As such, the dynamic binary instrumentation tool knows that the code executed in these address ranges needs to be disregarded.

Another issue is that many virtual machines use the notion of absolute time to trigger various internal events. This could be the case for detecting hot code that needs to be scheduled for optimization. Detecting hot code can be done by sampling the call stack; when the number of samples of a given method gets above a given threshold, the method is considered for optimization. The Java thread scheduling also relies on the notion of time. Java threads get time quanta for execution and when a time quantum has finished, another Java thread can be scheduled. Running a virtual machine within a Javana system causes the virtual machine to run slower, and by consequence, this affects timer-based virtual machine events such as code optimization and Java thread scheduling. This can be solved by using deterministic replay techniques [3]. In fact, it is common practice in virtual machine research to solve the code optimization non-determinism by having the virtual machine write out its recompilation strategy during an uninstrumented run, and then reuse this recompilation strategy during the instrumented run.

2.6 A proof-of-concept Javana system

The Javana system is a general framework for building a system for building customized Java program analysis tools. Any virtual machine could be employed in this framework and any dynamic binary instrumentation tool could be used as well. In our experimental framework, we use the Jikes RVM as our virtual machine and we use DIOTA as our dynamic binary instrumentation tool.

2.6.1 Jikes RVM

The Jikes Research Virtual Machine [1] is an open source Java virtual machine written almost entirely in Java. Jikes RVM uses a compilation-only scheme (no interpretation) for translating Java bytecodes to native machine instructions. In our experiments we use the *FastAdaptive* profile: all methods are initially compiled using a baseline compiler, and hot methods are recompiled using an optimizing compiler.

Making the Jikes RVM Javana-enabled was easy. We only had to insert around two hundred lines of code (including comments) into the virtual machine in order to trigger the events intercepted by the dynamic binary instrumentation tool. More specifically, we added an event to the class loader, to the object allocator, to all garbage collectors when an object or code is being moved or deleted, to all compilers and optimizers when a method is being compiled or optimized, and to the thread management system when a thread is created, switched or terminated.

There is one peculiarity with instrumenting the Jikes RVM itself that needs special attention — this is because the Jikes RVM is written in Java. Instrumentation cannot be activated until the virtual machine is properly booted. This means that there are some virtual machine methods and objects that cannot be communicated to the binary instrumentation tool during virtual machine startup. This can be solved by communicating these virtual machine methods and objects as soon as the virtual machine is properly booted. From then on, the instrumentation tool intercepts all method calls and object accesses during the program execution.

2.6.2 DIOTA

The dynamic binary instrumentation tool that we use in our proof-of-concept Javana system is DIOTA [17]. DIOTA stands for Dynamic Instrumentation, Optimization and Transformation of Applications and is a dynamic binary instrumentation framework for use on the Linux operating system running on x86-compatible processors. Its functionality includes intercepting memory operations, code execution, signals, system calls and functions based on their name or address, as well as the ability to instrument self-modifying code [16].

DIOTA is implemented as a dynamic shared library that can be hooked up to any program. The main library of DIOTA contains a generic dynamic binary instrumentation infrastructure. This generic instrumentation framework can be used by so-called backends that specify the particular instrumentation of interest that needs to be done. The backend that we currently use is a memory operation tracing backend, *i.e.*, this backend instruments all memory operations.

The general operation of DIOTA is very similar to that of other dynamic binary instrumentation frameworks such as PIN [15] and Valgrind [18]. All of these operate in a similar way as described in section 2.2.

3. The Javana language

A system for building customized Java program analysis tools also requires an easy-to-use instrumentation framework. The instrumentation framework is the environment in which the end user will build its profiling tools. In this paper, we introduce the Javana instrumentation language for building Java program profiling tools. The Javana instrumentation language is inspired by the Aspect-Oriented Programming (AOP) paradigm because AOP matches the needs in instrumentation very well.

time_qualifier := `before` | `after`
params := `location_t const_t *`*loc* `,` `type_t const *`*type* `,` `void **`*userdata*
object_operation := `create` (params) | `copy` (params, params) | `delete` (params)
object_event := `object` **:** object_operation
memory_operation := `read` (params) | `write` (params) | `access` (params)
memory_operation_target := `object` | `nonobject` | `any`
memory_event := memory_operation_target **:** memory_operation
event := time_qualifier memory_event | object_event {*advice code*}

Figure 2. The grammar of the Javana instrumentation language.

3.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [13] is best known in the context of high-level languages and software design methodologies, ranging from UML [24] and AspectJ for Java [12] to AspectC++ for C++ [22] to TinyC^2 for C [26]. The basic idea of aspect-oriented programming originally came from the observation that not all functionality in a programming model can be cleanly separated into objects or modules. Some requirements crosscut entire class hierarchies, multiple modules and complete programs. Aspect-oriented programming allows for specifying a desired functionality that concerns the whole program in a modular implementation.

Logging an application's execution is one of the best known examples. Implementing a logging facility in a traditional manner without AOP requires that logging code is inserted all over the program. This is very time-consuming, error-prone and hard to maintain from a software development point of view. AOP on the other hand allows for extracting this logging facility into a separate module, that is then *woven* by a *weaver* with the rest of the program at compile time or even at run time. AOP thus significantly improves software maintainability.

In general, an AOP language consists of *joinpoints*, *pointcuts* and finally the *advice*. A joinpoint specifies where and when one can interfere in the structure or execution of a program. This can range from source code line numbers to syntactical constructions to even run time events. A pointcut is a collection of joinpoints. Typically, a symbolic name can be associated with a pointcut for later reference. Finally, the advice is code that is associated with a pointcut. The advice will be executed whenever the conditions specified by the pointcut are fulfilled.

The general idea of AOP languages of segregating crosscutting concerns in separate modules is also very much applicable to the low-level instrumentation of programs at the machine code level. In fact, instrumenting a binary involves inserting additional code across the entire program in order to measure a program metric of interest [4, 14, 15, 18, 23]. Since the instrumentation itself can be completely segregated from the original program, AOP is a natural way for specifying instrumentation routines [19]. The Javana instrumentation language is inspired by the AOP concept.

3.2 The Javana instrumentation language

The Javana instrumentation language is a domain-specific language developed for the purpose of instrumenting programs written in object-oriented languages such as Java. It combines support for recognizing native execution information along with high-level language concepts such as objects, object types, methods, lines of code, threads, *etc.*

The grammar of the Javana instrumentation language is shown in Figure 2. A joinpoint that describes an event in the Javana instrumentation language consists of a time qualifier followed by a memory event or an object event, followed by the advice code. The time qualifier specifies when the event should be triggered. This can be before or after the event of interest. The events that can be triggered are memory events or object events. For each of those, a number of parameters are given. These parameters can then be used by the advice code. The advice code is the instrumentation code in C inserted by the end user.

```
struct mem_access_t {
  int ip;            /* instruction pointer */
  int addr;          /* memory address being accessed*/
  int size;          /* number of bytes accessed */
  int ld_st;         /* load or store ? */
  int thread_ID;     /* thread ID */
}

struct location_t {
  struct mem_access_t *ma;  /* pointer to
                               mem_access_t structure */
  int method_ID;     /* method ID */
  char* method_name; /* method's name */
  int line_number;   /* line number in given method */
}

struct type_t {
  int type_ID;       /* object class ID */
  char* type_name;   /* object class name */
}
```

Figure 3. Data structures provided in the Javana instrumentation language.

Javana also comes with a translator for converting the Javana instrumentation statements as specified in Figure 2 into C-statements while keeping the advice code (that is written in C) untouched. The translated instrumentation specification is then linked with DIOTA and the Jikes RVM for driving the profiling run.

We now discuss the object and memory events, the parameters that are provided with these events and finally the Javana directives. Example instrumentation specifications clarifying how to use the Javana instrumentation language in practice will be given in section 5.

3.2.1 Object events

An object event consists of the keyword `object` followed by an object operation. The object operation can be the creation (`create`), copying (`copy`) or deletion (`delete`) of an object.

3.2.2 Memory events

A memory event consists of memory operation target and the memory operation itself. The memory operation target can be an object, memory not belonging to an object or any of those. This allows the end user to focus the instrumentation of memory accesses to objects only, non-objects only, or to both objects and non-objects. The memory operation specifies the type of memory access that should be instrumented. This allows the end user to focus on reads, writes or both.

3.2.3 Parameters

The parameters that are provided by the Javana instrumentation language are shown in Figure 3. These parameters can be used in the advice code for driving the instrumentation. The first parameter is a data structure that collects information concerning the 'location' of the object or memory event. This is done in the `location_t` structure. The first element in this structure is a pointer to a `mem_access_t` structure. This latter structure contains (i) the instruction pointer of the native instruction performing the object or memory operation, (ii) the object's memory location or in case of a memory operation, the memory location being accessed, (iii) the size of the object or in case of a memory operation, the number of bytes accessed in memory, (iv) whether this memory access is a load or store operation—note this has no meaning in case of an object operation, and finally (v) the thread ID of the thread performing the object or memory operation. The second and third element in the `location_t` data structure are the method ID and the method name performing the object or memory operation, respectively. The fourth and final element is the source code line number in the given method that corresponds to this object or memory operation.

The second parameter in the parameter list is a pointer to a data structure that specifies information concerning the 'type' of the object or memory operation. This `type_t` structure holds a type ID and a type name of the object or memory operation. This means that for every object being created, copied, deleted or accessed, the Javana instrumentation language provides the end user with information concerning the object's type.

The third parameter in the parameter list (`void **`*userdata*) allows the end user to maintain object-specific information. The end user may for example set up a data structure for a given object; the pointer to this data structure can be stored through this third parameter. The binary instrumentation tool then makes sure that this pointer is available for all object and memory operations that refer to that same object.

3.2.4 Javana directives

The Javana instrumentation language also comes with a number of directives that can be specified at the beginning of the instrumentation specification file. There are two directives in our current implementation, namely `#pragma requires method_info` and `#pragma requires object_info`. The purpose of these directives is to improve performance, *i.e.*, to reduce the overhead of the vertical instrumentation. The `#pragma requires method_info` directive informs the dynamic binary instrumentation tool that the method ID, the method name and source code line number should be kept track of during binary instrumentation. The `#pragma requires object_info` directive informs the dynamic binary instrumentation tool that the object type ID and the object type name should be kept track of.

The user can decide not to include any of these two directives in the instrumentation specification, to include only one of these, or to include both. This will affect the amount of information that can be gathered during a profiling run as well as the amount of overhead experienced during profiling. For example, if a user is interested in measuring the cache miss rate per method and per source code line number, then there is no benefit in collecting per-object information. The user can then use the `#pragma requires method_info` to disable tracking object-related information during the instrumentation run. This will limit the slowdown during vertical profiling.

4. Javana performance

This section quantifies the slowdown of the Javana system.

4.1 Experimental setup

In our evaluation of Javana's performance we use the SPECjvm98 benchmark suite, the SPECjbb2000 benchmark as well as the DaCapo benchmarks, see Table 1. The SPECjvm98 benchmark suite[1] is a client-side Java benchmark suite consisting of seven benchmarks. We run all SPECjvm98 benchmarks with the largest input set (`-s100`). SPECjbb2000[2] emulates the middle-tier of a three-tier system; the pseudojbb variant of the SPECjbb2000 benchmark that we use in our analysis runs for a fixed amount of work, *i.e.*, for a fixed number of transactions, in contrast to SPECjbb2000 which runs for a fixed amount of time. The DaCapo benchmark suite[3] is an open-source benchmark suite designed for memory management research; we use release version beta050224. All of the SPECjvm98 benchmarks are run on the Jikes RVM using a 64MB heap and the generational mark-sweep (GenMS) garbage collector; pseudojbb and the DaCapo benchmarks are run with a 500MB heap. Our measurements are done on a 2.8GHz Intel Pentium 4 system with a 512KB L2 cache and 1GB main memory. The operating system on which we run our experiments is Gentoo Linux with a 2.6.10 kernel.

4.2 Javana overhead analysis

Running a Java application within Javana obviously introduces overhead. There are a number of contributors to the overall overhead:

- First, the dynamic binary instrumentation tool that runs underneath the virtual machine causes overhead.
- Second, the event handling mechanism that communicates high-level language concepts from the virtual machine to the dynamic binary instrumentation tool also causes overhead. In addition, the event handler needs to process this information for updating the vertical map in the dynamic binary instrumentation tool.
- Third, executing instrumented code requires that the binary instrumentation tool searches the vertical map for every memory location accessed.
- And finally, executing the instrumentation code itself as implemented by the end user of the Javana system also causes additional overhead.

We will now quantify the overhead caused by each of these four overhead contributors.

4.2.1 Dynamic binary instrumentation overhead

We first quantify the overhead of the binary instrumentation (bullet one from above). There are two contributors to this overhead. First, whenever a control transfer occurs to a computed address, this address must be looked up in the binary instrumentation engine in order to transfer control to the corresponding instrumented code. The overhead that we observe for DIOTA in our Javana system ranges from 1.5X to 5.5X, see Figure 4.

The second contributor is due to calling an instrumentation routine for all natively executed memory operations. We quantify this overhead by calling a dummy (empty) function for each memory operation. The overhead varies between a factor 12X and 53X depending on the benchmark, see Figure 4. This overhead is inherent to dynamic binary instrumentation. Other dynamic binary instrumentation tools such as PIN [15] and Valgrind [18] show similar overheads.

4.2.2 Vertical instrumentation overhead

We now quantify the overhead caused by the event handling mechanism and by searching the vertical map for every memory location

[1] `http://www.spec.org/jvm98/`

[2] `http://www.spec.org/jbb2000/`

[3] `http://www-ali.cs.umass.edu/DaCapo/gcbm.html`

Suite	Benchmark	Description
SPECjvm98	jess	Solves a number of puzzles with varying degrees of complexity
	db	Performs a set of database requests on a memory resident database
	javac	Compiles a number of Java files
	mpegaudio	Decompresses MPEG I Layer 3 audio files
	mtrt	Renders a scene using ray tracing
	jack	Parses grammar files and generates a parser for each
SPECjbb2000	pseudojbb	Emulates the middle tier of a three tier system
DaCapo	hsqldb	Executes a number of transactions against a memory resident database
	antlr	Parses grammar files and generates a parser and lexical analyzer for each of them
	fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file
	jython	Interprets a series of Python programs
	ps	Reads and interprets a PostScript file
	xalan	Transforms XML documents into HTML

Table 1. The benchmarks used in this paper.

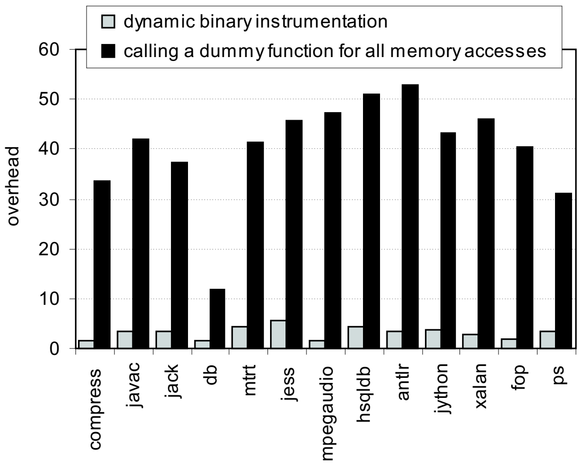

Figure 4. Slowdown due to dynamic binary instrumentation.

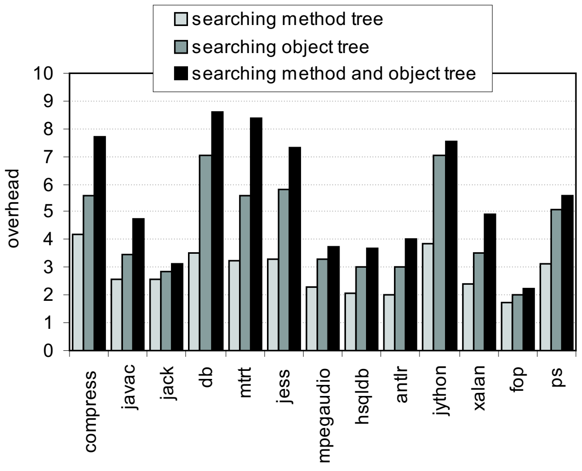

Figure 5. Slowdown due to vertical instrumentation.

accessed (bullets two and three from above). We collectively refer to this overhead as vertical instrumentation overhead, *i.e.*, this is the overhead that enables cross-layer instrumentation. In our experiments we observed that the event handling mechanism is only a very small part of the total vertical instrumentation overhead.

Figure 5 quantifies the overhead from vertical instrumentation.

- The first bar for each benchmark shows the overhead for the Javana system during a vertical profiling run that only considers method-related information, *i.e.*, the Javana directive `#pragma requires method_info` was set in the instrumentation specification. This causes the method tree to be searched for every memory access; the object tree is not searched.
- The second bar measures the overhead when enabling vertical profiling for measuring object-related information. The Javana directive `#pragma requires object_info` was set. In other words, the vertical object tree is searched, but not the method tree.
- The third bar quantifies the overhead when both the method tree and the object tree are searched. Both the `#pragma requires method_info` and `#pragma requires object_info` directives are set.

The average vertical instrumentation overhead varies between a factor 2.8X and 5.5X depending on what information is to be kept track of. The third bar (searching the method and object trees) quantifies the total overhead; and this causes an average slowdown of a factor 5.5X. However, using the Javana directives, see two leftmost bars 'searching method tree' and 'searching object tree', significant reductions in overhead are obtained. The average slowdown for the vertical method and object instrumentation is a factor 2.8X and 4.4X, respectively.

4.2.3 Overall overhead

From the above enumeration, it follows that the total slowdown of a Java program analysis tool built within Javana equals the product of the dynamic binary instrumentation slowdown, the vertical instrumentation slowdown and the slowdown due to the user-defined instrumentation routines, *i.e.*, the advice code included in the instrumentation specification.

The total slowdown for the dynamic binary instrumentation and the vertical instrumentation varies between a factor 90X and 345X. This is the overhead caused by using Javana. The additional overhead due to the instrumentation routines, increases the overall overhead to the 125X-850X range; this is for the vertical cache simulation which is the most demanding vertical profiling tool that we built with Javana.

According to our experience, this is an acceptable slowdown. Compared to simulation, Javana is fast; simulation typically causes a slowdown by at least a factor 10,000X [2]. In cases where a 90X to 345X slowdown is undesirable, sampling can be employed

```
before any:access (location_t const *loc, type_t const *type, void **userdata) {
  printf ("access by insn @ %p to memory location %p of size %d\n", loc->ma->ip, loc->ma->addr, loc->ma->size);
}
```

Figure 6. Memory address tracing tool in Javana.

to reduce this slowdown. However, this comes at the price of lost accuracy; our measurements were done without applying any sampling.

5. Applications

We now discuss three example applications of the Javana system: memory address trace generation, vertical cache simulation and object lifetime computation. These applications demonstrate the real power of Javana: Javana provides an easy-to-use instrumentation environment that allows for quickly building customized (vertical) Java program analysis tools. The key benefit is that easy-to-build program analysis tools increase a software designer's productivity. And in addition, the results from profiling runs could yield invaluable information for optimizing the application.

5.1 Memory address trace generation

Our first application is memory access tracing; the instrumentation specification for building this profiling tool is shown in Figure 6. This profiling tool captures all memory accesses during program execution and writes to a file each access' instruction pointer, memory address and size. As can be seen from Figure 6, the Javana instrumentation language only requires three lines of code for building this profiling tool. In other words, the expressiveness of the Javana language is high while the code itself is very intuitive.

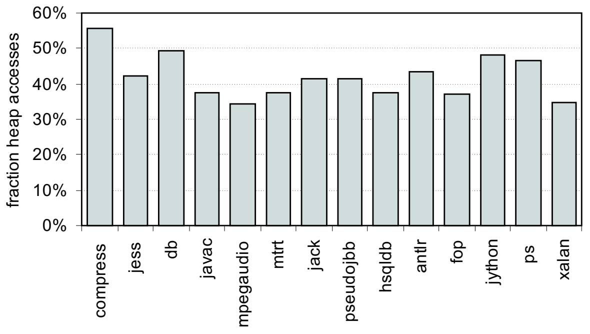

Figure 7. The fraction of all memory accesses that are heap accesses.

Recent work done by Shuf *et al.* [21] analyzed the memory behavior of Java applications. For doing so, Shuf *et al.* modified the virtual machine to trace all accesses to the heap, however, they did not trace accesses to the stack — presumably because it is very difficult to track all memory accesses including stack accesses by instrumenting the virtual machine. Using Javana we built a profiling tool that traces all heap memory accesses and all stack memory accesses. We found that on average only 42% of all memory accesses are heap accesses, see Figure 7 which shows the fraction heap accesses compared to the total number of data memory accesses. In other words, Shuf *et al.* captured only 42% of the total number of memory accesses on average. Consequently, not capturing the large fraction of non-heap accesses has a significant impact on the overall memory system behavior. Figure 8 shows the fraction L1 and L2 misses as a ratio to the total number of memory accesses. These results are for a simulated 4-way set-associative 32KB 32-byte line L1 cache and an 8-way set-associative 1MB 128-byte line L2 cache. Both are write-back, write-allocate caches. The cache simulation routines were taken from the SimpleScalar Tool Set [2]. We observe that only considering heap accesses results in a severe overestimation of the actual cache miss rates. The difference in miss rates varies by a factor 1.8 and 2.9 between tracking heap accesses versus tracking all memory accesses. Therefore, we conclude that a methodology that analyzes heap accesses only in a memory performance study, is questionable.

5.2 Vertical cache simulation

The second application relates cache miss rates to high-level concepts such as methods, source code lines, objects and object types. This is invaluable information for software developers that are in the process of optimizing their code for memory performance. As is well known, the memory-processor speed gap is an important issue in current computer systems. Poor memory behavior can severely affect overall performance. As such, it is very important to optimize memory performance as much as possible. Vertical profiling is a very valuable tool for hinting the software developer what to focus on when optimizing the application's memory behavior.

Vertical cache simulation requires that an instrumentation specification be written as shown in Figure 9. Lines 0 and 1 specify that the instrumentation needs to keep track of both per-object and per-method information. Upon a memory access to an object (lines 2-6), the memory address is used by the cache simulator to update the cache's state. The type-specific and method-specific data structures maintained by the instrumentation tool are updated to keep track of the per-type and per-method miss rates. Other memory accesses, *i.e.*, to non-objects (lines 7-10), update the cache's state and update the per-method miss rate information. The per-type miss rate information is not updated because these memory references do not originate from object accesses.

The instrumentation specification for this profiling tool was no more than 200 lines of code, including comments. The output of the profiling run is a table describing cache miss rates per method, per line of code, per object and per object type.

Selecting the per-method and per-object type cache miss rates and sorting them by decreasing number of L2 misses results in Tables 2 and 3. In both tables we limit the number of methods and object types to the top five per benchmark in order not to overload the tables. The first column in each table mentions the method or object type, respectively. The second column shows the percentage of memory references of the given method or object type as a percentage of the total number of memory references. The two rightmost columns show the number of L1 and L2 misses, respectively, along with the percentage local miss rates, *i.e.*, the number of misses divided by the number of accesses to the given cache level.

Software developers can use these tables to better understand the memory behavior of their software for guiding memory optimizations at the source code level. For example, from Table 2 it is apparent that the `shell_sort` method in db is a method that suffers heavily from poor cache behavior. About 60% of the memory references in db occur within the `shell_sort` method. Of these memory references, 10.5% result in an L1 cache miss, and 31.7% of the L2 cache accesses are cache misses. As such, this method is definitely a method of concern to a software developer when optimizing the memory performance of db.

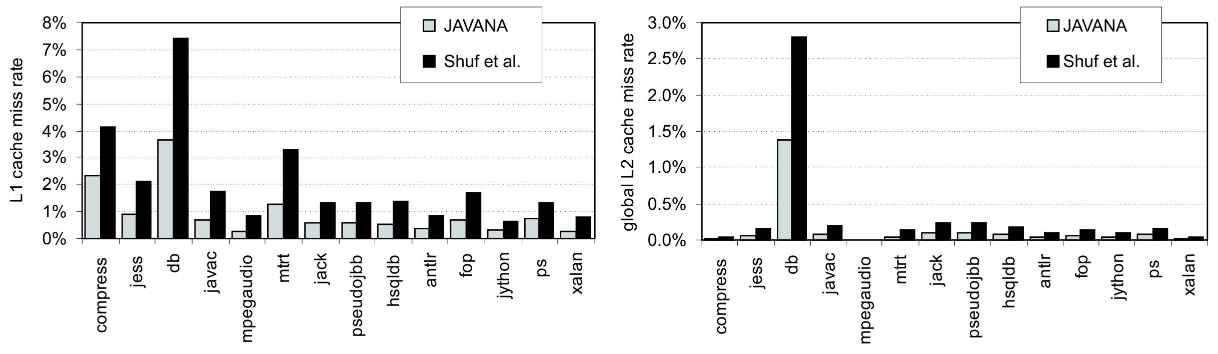

Figure 8. Cache miss rates using Javana versus Shuf *et al.*'s methodology: L1 data cache miss rates (number of L1 data cache misses divided by the number of L1 data accesses) on the left and global L2 data cache miss rates (number of L2 data misses divided by the number of L1 data accesses) on the right.

```
#pragma requires object_info
#pragma requires method_info

before object:access (location_t const *loc, type_t const *type, void **userdata) {
  /* compute whether this object reference is a cache miss or not */
  hit = simulate_memory_access (loc->ma->addr, type->type_ID);
  /* update the per-type hit/miss information */
  update_per_type_miss_rate (type->type_ID, hit);
  update_per_method_miss_rate (loc->method_name, loc->line_number);
}

before nonobject:access (location_t const *loc, type_t const *type, void **userdata) {
  /* update the simulated cache content */
  simulate_memory_access (loc->ma->addr, -1);
  update_per_method_miss_rate (loc->method_name, loc->line_number);
 }
```

Figure 9. Vertical cache simulation tool in Javana.

```
#pragma requires object_info

typedef {
  unsigned long long creation_time;
  unsigned long long last_access;
} object_info_t;

static unsigned long long timestamp = 0;

after object:create (location_t const *loc, type_t const *type, void  **userdata) {
  object_info_t ** const objectinfo = (object_info_t**)userdata;

  (*objectinfo) = diota_malloc(sizeof(object_info_t));
  (*objectinfo)->creation_time = timestamp;
  (*objectinfo)->last_access = 0;
}

before object:access (location_t const *loc, type_t const *type, void  **userdata) {
  object_info_t ** const objectinfo = (object_info_t**)userdata;

  timestamp++;
  (*objectinfo)->last_access = timestamp;
}

before nonobject:access (location_t const *loc, type_t const *type,  void **userdata) {
  timestamp++;
}
```

Figure 10. Object lifetime computation tool in Javana.

Method	Accesses	DL1 misses	DL2 misses
_201_compress			
`Compressor.compress()V`	42.7%	130906173 (7.8%)	533099 (0.4%)
`Decompressor.decompress()V`	42.7%	21390490 (1.3%)	485995 (2%)
`Input_Buffer.readbytes([BI)I`	1.8%	247545 (0.3%)	55151 (18.4%)
`Compressor.output(I)V`	4.5%	207706 (0.1%)	35700 (14%)
`Output_Buffer.putbyte(B)V`	0.9%	125551 (0.3%)	25169 (17.2%)
_213_javac			
`Assembler.add(IILjava/lang/Object;)V`	0.3%	188725 (3.2%)	46040 (16.9%)
`CompoundStatement.check(LEnvironment;LContext;JLjava/util/Hashtable;)J`	0.1%	184186 (12%)	32186 (13.2%)
`Expression.<init>(IILType;)V`	0.2%	138495 (2.8%)	29645 (15.1%)
`Instruction.<init>(IILjava/lang/Object;)V`	0.1%	115000 (4.6%)	27981 (16.8%)
`CompoundStatement.code(LEnvironment;LContext;LAssembler;)V`	0.1%	107313 (7.3%)	27416 (17.7%)
_228_jack			
`TokenEngine.getNextTokenFromStream()LToken;`	4%	214496 (0.5%)	8757 (3.3%)
`Token.<init>()V`	0.1%	31771 (2.6%)	7689 (14.1%)
`JackConstants.printToken(LToken;Ljava/io/PrintStream;)V`	0.2%	18382 (1%)	4564 (14.1%)
`TokenProcessor.action(LExpansion;)V`	0%	5299 (2.5%)	2568 (32.7%)
`RunTimeNfaState.Move(CLjava/util/Vector;)I`	4.5%	369592 (0.7%)	2168 (0.4%)
_209_db			
`Database.shell_sort(I)V`	59.8%	132442434 (10.5%)	50720398 (31.6%)
`Entry.equals(Ljava/lang/Object;)Z`	3.5%	3413385 (4.6%)	1720280 (36%)
`Database.set_index()V`	6%	3924453 (3.1%)	1345881 (28.7%)
`Database.read_db(Ljava/lang/String;)V`	0.8%	36682 (0.2%)	9152 (13%)
`spec.io.FileInputStream.read()I`	0%	5078 (0.7%)	3927 (60%)
_227_mtrt			
`OctNode.FindTreeNode(LPoint;)LOctNode;`	11.8%	27446406 (12.1%)	463179 (1.5%)
`PolyTypeObj.Intersect(LRay;LIntersectPt;)Z`	3.9%	1718595 (2.3%)	184134 (9.9%)
`Vector.<init>(FFF)V`	0.3%	715338 (13.7%)	177453 (22%)
`OctNode.Intersect(LRay;LPoint;F)LOctNode;`	16.4%	1463021 (0.5%)	145254 (8.9%)
`Face.GetVert(I)LPoint;`	14.3%	9920823 (3.6%)	113315 (1%)
_202_jess			
`jess.Node2.appendToken(Ljess/Token;Ljess/Token;)Ljess/Token;`	5.4%	6482818 (7.5%)	490280 (5.7%)
`jess.Value.<init>(DI)V`	0.9%	786943 (5.3%)	176391 (16.5%)
`jess.RU.getAtom(I)Ljava/lang/String;`	2.1%	724295 (2.1%)	115008 (12%)
`jess.Node2.findInMemory(Ljess/TokenVector;Ljess/Token;)Ljess/Token;`	1.7%	3197035 (11.5%)	26021 (0.6%)
`jess.Value.<init>(II)V`	0.1%	120645 (8.2%)	20881 (13.1%)
hsqldb			
`Column.createSQLString(Ljava/lang/Object;I)Ljava/lang/String;`	0.6%	490464 (3%)	353687 (44.4%)
`Table.getInsertStatement([Ljava/lang/Object;)Ljava/lang/String;`	0.8%	325886 (1.4%)	145450 (27.7%)
`Index.next(LNode;)LNode;`	0.2%	152695 (3.3%)	141248 (53.8%)
`Expression.<init>(ILjava/lang/Object;)V`	0.2%	545316 (9.6%)	135686 (14.3%)
`Result.<init>()V`	0%	142437 (10%)	35338 (14.3%)
antlr			
`collections.impl.BitSet.toArray()[I`	2.8%	678978 (1.2%)	168854 (13.6%)
`collections.impl.BitSet.orInPlace(Lcollections/impl/BitSet;)V`	1.1%	440776 (2%)	92577 (13%)
`Lookahead.<init>()V`	0.1%	48837 (4.7%)	10240 (13.5%)
`AlternativeBlock.getAlternativeAt(I)LAlternative;`	0.3%	73447 (1.2%)	8135 (7.5%)
`LLkAnalyzer.look(ILAlternativeBlock;)LLookahead;`	0.1%	76247 (4.4%)	7967 (7%)
jython			
`core.Py.newInteger(I)Lcore/PyInteger;`	2.3%	3046300 (2.1%)	677551 (13.5%)
`pycode._pyx2.mmult$2(Lcore/PyFrame;)Lcore/PyObject;`	7.2%	2688617 (0.6%)	367581 (8.6%)
`core.PyObject.__iadd__(Lcore/PyObject;)Lcore/PyObject;`	1.3%	1414629 (1.8%)	348087 (16%)
`core.PyObject._add(Lcore/PyObject;)Lcore/PyObject;`	1%	568570 (0.9%)	139698 (15.9%)
`core.PySequence.__finditem__(I)Lcore/PyObject;`	3.4%	470246 (0.2%)	114780 (14.1%)
xalan			
`utils.SuballocatedIntVector.addElement(I)V`	6.1%	1178740 (0.7%)	230876 (11.8%)
`utils.SuballocatedIntVector.elementAt(I)I`	1.4%	648314 (1.7%)	79537 (10.2%)
`dtm.ref.DTMDefaultBase.indexNode(II)V`	0.4%	77218 (0.7%)	11836 (9.3%)
`dtm.ref.DTMDefaultBase.findGTE([IIII)I`	1.4%	96575 (0.2%)	8676 (7.2%)
`serializer.WriterToUTF8Buffered.write([CII)V`	0.1%	23178 (1.2%)	1307 (4.5%)
fop			
`fo.flow.Block.layout(Llayout/Area;)I`	0.1%	116716 (18.9%)	3900 (2.7%)
`layout.inline.InlineSpace.<init>(I)V`	0%	12918 (8.1%)	3297 (20.6%)
`fo.FOText.layout(Llayout/Area;)I`	0%	60908 (18.3%)	3297 (4.4%)
`fo.expr.PropertyTokenizer.<init>(Ljava/lang/String;)V`	0%	14369 (6.5%)	3085 (15.4%)
`fo.PropertyList.get(Ljava/lang/String;ZZ)Lfo/Property;`	0.6%	108845 (2.4%)	2437 (1.8%)
ps			
`PSObject.DictionaryObject.getValueOf(LPSObject/PSObject;)LPSObject/PSObject;`	10.4%	6274191 (1.5%)	247961 (2.8%)
`State.DictStack.getValueOf(LPSObject/PSObject;)LPSObject/PSObject;`	1.2%	585068 (1.3%)	78663 (9.7%)
`PSObject.PSObject.<init>()V`	0.1%	126756 (5.1%)	12993 (7.9%)
`PSObject.ProcedureObject.execute()V`	0.5%	867436 (4%)	3130 (0.2%)
`State.PathPoint.<init>(DD)V`	0%	22820 (6.4%)	3124 (9.9%)

Table 2. The top 5 methods for each of the benchmarks sorted by the number of L2 cache misses.

Type	Accesses	DL1 misses	DL2 misses
_201_compress			
`[B`	20.6%	13001417 (1.6%)	1038857 (7.1%)
`[I`	9.2%	100254792 (27.8%)	56465 (0%)
`[S`	4.7%	41699357 (22.6%)	54699 (0.1%)
`[Ljava/lang/Object;`	0.4%	4010 (0%)	406 (8.8%)
`[[I`	0.1%	1632 (0%)	368 (19.6%)
_213_javac			
`LInstruction;`	1%	1809398 (7.8%)	95228 (3.6%)
`LFieldExpression;`	0.3%	297830 (4.2%)	54787 (13.1%)
`LIdentifierExpression;`	0.3%	376774 (5.1%)	51935 (9.8%)
`LExpressionStatement;`	0.1%	179932 (7.2%)	34840 (13.9%)
`LMethodExpression;`	0.2%	195649 (3.8%)	30122 (11%)
_228_jack			
`LToken;`	0.2%	96473 (3.6%)	19971 (12.3%)
`[I`	4.1%	297878 (0.6%)	3900 (0.8%)
`[J`	0.7%	18526 (0.2%)	2600 (8.7%)
`Ljava/util/Hashtable;`	1.2%	130877 (0.9%)	2389 (1.1%)
`Ljava/lang/String;`	2.5%	23519 (0.1%)	2384 (7%)
_209_db			
`Ljava/util/Vector;`	15.3%	36375367 (11.2%)	17288803 (38.8%)
`[Ljava/lang/Object;`	7.2%	24118101 (15.7%)	11838596 (41.3%)
`[C`	11.7%	22697725 (9.1%)	11598229 (42.4%)
`LEntry;`	4.1%	28511936 (32.6%)	7941652 (22.7%)
`Ljava/lang/String;`	13%	22717143 (8.3%)	4348649 (15.6%)
_227_mtrt			
`LVector;`	5.5%	3968361 (3.7%)	554763 (12.7%)
`LPoint;`	10.6%	15020935 (7.4%)	358453 (2.1%)
`[LPoint;`	3.5%	10720458 (16%)	114210 (1%)
`[I`	4.7%	1055491 (1.2%)	97335 (8.4%)
`LFace;`	3.3%	6796479 (10.7%)	82116 (1.1%)
_202_jess			
`[Ljess/ValueVector;`	6.7%	5644607 (5.3%)	370383 (5%)
`Ljess/Value;`	4.5%	3643209 (5.1%)	216290 (4.6%)
`Ljava/lang/Integer;`	0.5%	374152 (4.4%)	80287 (15.8%)
`Ljess/Token;`	4.5%	5324507 (7.4%)	43438 (0.6%)
`[Ljess/Value;`	6.2%	3964254 (4%)	13348 (0.3%)
hsqldb			
`Ljava/lang/Integer;`	0.6%	443743 (2.5%)	353576 (47.9%)
`LMemoryNode;`	0.8%	1098884 (5%)	174004 (10.8%)
`[Ljava/lang/Object;`	2.5%	2085372 (2.9%)	147628 (5.1%)
`LExpression;`	0.3%	685749 (9.3%)	135919 (11.4%)
`LResult;`	0%	142438 (23.7%)	35359 (14.3%)
antlr			
`[I`	6.7%	2250997 (1.7%)	201101 (6.3%)
`[J`	3.7%	677296 (0.9%)	114003 (10.7%)
`[C`	5.5%	95725 (0.1%)	13275 (9%)
`LLookahead;`	0.2%	143657 (4.6%)	12755 (6%)
`LAlternative;`	0.1%	137362 (6.3%)	10652 (5.2%)
jython			
`Lcore/PyInteger;`	4.4%	8793859 (3.3%)	1424117 (9.9%)
`Lcore/PyList;`	5.8%	254839 (0.1%)	22977 (6.4%)
`[Lcore/PyObject;`	5%	3302410 (1.1%)	13465 (0.2%)
`[I`	2.8%	81657 (0%)	7148 (6.1%)
`[Ljava/lang/Object;`	8%	71723 (0%)	5139 (4.8%)
xalan			
`[I`	22.3%	3588258 (0.6%)	332505 (5.9%)
`[B`	4.2%	506966 (0.4%)	2158 (0.3%)
`[[I`	1.9%	350386 (0.7%)	1375 (0.3%)
`[Ljava/lang/Object;`	2.7%	447896 (0.6%)	1062 (0.1%)
`[[C`	0.1%	36942 (1.2%)	1055 (1.7%)
fop			
`[I`	7.8%	902018 (1.5%)	9785 (0.9%)
`Lfo/PropertyList;`	1.2%	68644 (0.7%)	5392 (6.3%)
`[C`	4.1%	88207 (0.3%)	5361 (4.4%)
`[Ljava/lang/Object;`	2.5%	752290 (3.7%)	5205 (0.6%)
`Llayout/inline/InlineSpace;`	0%	25860 (11.2%)	3592 (9.9%)
ps			
`LExceptions/PSObjectException;`	0.1%	350039 (6.7%)	86914 (15.5%)
`LPSObject/realObject;`	0.1%	150678 (7%)	9650 (4.2%)
`LState/PathPoint;`	0%	26141 (5%)	3162 (8.2%)
`Ljava/lang/String;`	1.9%	2434633 (3.2%)	2492 (0.1%)
`LPSObject/NullObject;`	0%	7604 (12.4%)	1919 (18%)

Table 3. The top 5 objects types for each of the benchmarks sorted by the number of L2 cache misses.

	Source code	DL1 accesses	DL1 misses	DL2 accesses	DL2 misses
1	`void shell_sort(int fn) {`				
2	`  int i, j, n, gap;`				
3	`  String s1, s2;`				
4	`  Entry e;`				
5					
6	`  if (index == null) set_index();`	67	0 (0%)	0	0 (0%)
7	`  n = index.length;`	134	1 (0%)	1	0 (0%)
8					
9	`  for (gap = n/2; gap > 0; gap/=2)`	938	0 (0%)	0	(0%)
10	`    for (i = gap; i < n; i++)`	12276499	910 (0%)	1083	3 (0%)
11	`      for (j = i-gap; j >=0; j-=gap) {`	23064743	8179 (0%)	9615	33 (0%)
12	`        s1 = (String)index[j].items.elementAt(fn);`	157553557	29772665 (19%)	36551726	6095594 (17%)
13	`        s2 = (String)index[j+gap].items.elementAt(fn);`	157553557	24036992 (15%)	29456752	15581062 (53%)
14					
15	`        if (s1.compareTo(s2) <= 0) break;`	45015302	128 (0%)	153	1 (0%)
16					
17	`        e = index[j];`	32322537	219 (0%)	228	0 (0%)
18	`        index[j] = index[j+gap];`	75419253	2654 (0%)	3228	811 (25%)
19	`        index[j+gap] = e;`	43096716	0 (0%)	0	0 (0%)
20	`      }`				
21	`  fnum = fn;`	67	61 (91%)	73	61 (84%)
22	`}`				

Table 4. The `shell_sort` method from db annotated with cache miss information. The number of L1 and L2 misses differ from the numbers given Table 2; the reason is that the numbers in this table were obtained using the baseline compiler whereas the numbers in Table 2 were obtained using the adaptive compiler; the line numbers returned by the adaptive compiler in Jikes are inaccurate.

Table 3 shows per-object type miss rates for the various benchmarks. The poor cache behavior for db seems to be apparent across a number of object types. For example, this table shows that the cache behavior for the `Vector` class is relatively poor with an L1 cache miss rate of 11.4% and an L2 miss rate of 38.6%. Note that our framework also allows for going even one step further, namely to tracking down miss rates to individual objects. This would allow the software developer to isolate the source of the poor memory behavior. We do not include an example of per-object miss rates here in this paper, however, this could be easily done in Javana.

Because the `shell_sort` method in db seems to suffer the most from poor cache behavior, we focus on that method now. Table 4 shows the `shell_sort` method annotated with cache miss information, *i.e.*, L1 and L2 cache miss rates are annotated to each line of source code. Line 13 seems to be the primary source for the high cache miss rate in the `shell_sort` method. The reason is that the `j+gap` index results in a fairly random access pattern into the 61KB `index` array. It's interesting to note that Hauswirth *et al.* [11] also identified the `shell_sort` method as a critical method for db.

5.3 Object lifetime

Our third example application computes object lifetimes. In this application we define the object lifetime as the number of memory accesses between the creation and the last use of an object. Knowing the allocation site and knowing where the object was last used can help a programmer to rewrite the code in order to reduce the memory consumption of the application or even improve overall performance [20].

Computing object lifetimes without Javana is fairly complicated. First, the virtual machine needs to be extended in order to store the per-object lifetime information. Second, special care needs to be taken so that the computed lifetimes do not get perturbed by the instrumentation code. Finally, all object references need to be traced. This is far from trivial to implement. For example, referencing the object's header is required for accessing the Type Information Block (TIB) or vertical lookup table (vtable) on a method call, for knowing the object's type, for knowing the array's length, *etc*. Also, accesses to objects in native methods need to be instrumented manually. Implementing all of this in a virtual machine is time consuming, error-prone and will likely be incomplete.

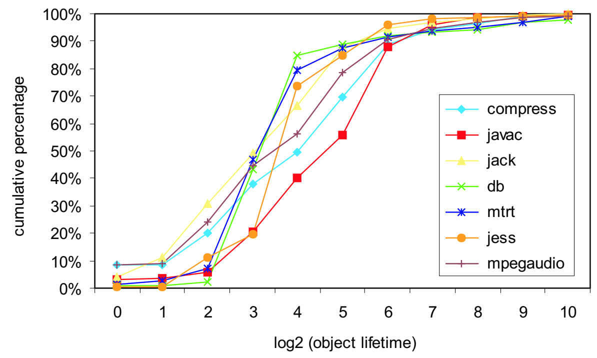

Figure 11. Cumulative object lifetime distribution for the SPECjvm98 benchmarks.

Measuring the object lifetime within Javana on the other hand is easy to do and in addition, it is accurate because it allows for tracking *all* references to a given object. In a Javana instrumentation specification, an object's lifetime can be computed and stored using the per-object `void **userdata` parameter that is available in Javana language, see section 3.2. As such, computing object lifetimes is straightforward to do in Javana — no more than 50 lines of code. The skeleton of the instrumentation specification is shown in Figure 10.

Figure 11 shows the lifetime distribution for the SPECjvm98 benchmarks computed using the Javana system. The horizontal axis on these graphs is given on a log_2 scale; the vertical axis shows the cumulative percentage objects in the given lifetime bucket. We observe that the object lifetimes are fairly small in general, *i.e.*, most objects are short-lived objects. For most benchmarks, the object lifetime typically is smaller than 16 memory accesses between the creation of an object and its last use. Some benchmark have a relatively larger object lifetime, see for example javac, compress and mpegaudio, however the object lifetime is still

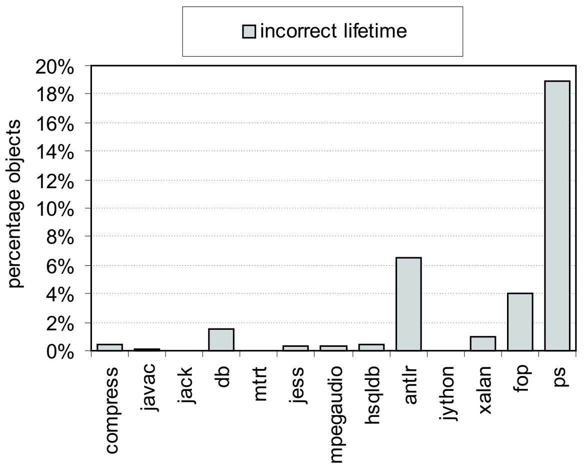

Figure 12. Evaluating the accuracy of object lifetime computations without Javana: the percentage objects for which a non-Javana instrumentation results in incorrect lifetime computations.

very small in absolute terms, *i.e.*, the object lifetime is rarely more than 64 memory accesses.

In order to evaluate the accuracy of object lifetime computations without Javana, we have set up the following experiment. We compute the object lifetimes under two scenarios. The first scenario computes the object lifetime when taking into account all memory accesses as done using out-of-the-box Javana. The second scenario computes the object lifetime while excluding all object accesses from non-Java code; this excludes all the object accesses from native JNI functions. This second scenario emulates current practice of building an object lifetime measurement tool within the virtual machine, without Javana. The results are shown in Figure 12. The graph shows the percentage of objects for which an incorrect lifetime is computed in current practice, *i.e.*, when not including accesses to objects through JNI functions. We observe large error percentages for a couple of benchmarks, namely fop (4%), antlr (6.5%) and ps (19%). As such, we conclude that current practice of computing object lifetime without Javana can yield incorrect results, and this could be misleading when optimizing the code based on these measurements.

6. Related work

We now discuss related work. We first discuss binary instrumentation tools followed by bytecode-level instrumentation approaches. Finally, we detail on existing vertical profiling approaches using hardware performance counters.

6.1 Binary instrumentation tools

A large body of work exists on instrumentation. A number of static instrumentation tools have been proposed such as ATOM [23], EEL [14] and FIT [4]. Static instrumentation tools take a binary and store an instrumented version of the binary on disk. Executing the instrumented binary then generates the desired profile information. Static instrumentation cannot be used for analyzing Java applications because it cannot deal with dynamically generated code.

Dynamic instrumentation on the other hand does not have that limitation. Well known examples of dynamic binary instrumentation frameworks are Valgrind [18], PIN [15] and DIOTA [16, 17].

One specific tool within Valgrind's tool set is Cachegrind which is a cache profiler that provides limited vertical profiling capabilities. However, Cachegrind is unable to vertically profile Java applications, nor is it capable of mapping cache miss rates to objects or object types.

6.2 Bytecode-level profiling

A number of Java bytecode-level profiling tools have been presented in the recent literature. These bytecode-level profiling tools differ from the Javana system in that Javana allows for building vertical profiling tools, whereas bytecode-level profiling tools instrument the intermediate bytecode level. We discuss two bytecode-level profiling tools now.

Dufour *et al.* [7] studied the dynamic behavior of Java applications in an architecture-independent way. To do so, they built a tool called *J [8] that uses the Java Virtual Machine Profiling Interface (JVMPI) to collect a wide set of bytecode-level Java program characteristics. The Java metrics that they collect are related to program size and structure, the occurrence of various data structures (such as arrays, pointers, *etc.*), polymorphism, memory usage, concurrency and synchronization.

Dmitriev [5] presents a Java bytecode-level profiling tool called JFluid. JFluid can be attached to a running Java application. The attached JFluid then injects instrumentation bytecodes into the methods of the running Java program. The instrumentation bytecodes collect profiling information online. When desired, JFluid can be detached from the running application.

6.3 Vertically profiling Java applications

Some very recent work focused on vertical profiling of Java applications. The purpose of these approaches is to link microprocessor performance to the Java application and the virtual machine. However, they do not allow for building customized vertical profiling tools.

Hauswirth *et al.* [11] and the earlier work by Sweeney *et al.* [25] presented a vertical profiling approach that correlates hardware performance counter values to manually inserted software monitors in order to keep track of the program's execution across all layers. The low-level and high-level information is collected at a fairly coarse granularity, *i.e.*, hardware performance counter values and software monitor values are measured at every thread switch. Hauswirth *et al.* measure various hardware performance metrics during multiple runs yielding multiple traces. And because of non-determinism during the execution, these traces subsequently need to be aligned. Although being much faster than Javana, there are two important limitations with this approach. First, aligning traces is challenging and caution is required in order not to get out of sync [10]. Second, the granularity is very coarse-grained — one performance number per thread switch. This allows for analyzing coarse-grained performance variations but does not allow for analyzing the fine-grained performance issues we target.

Georges *et al.* [9] also provided a limited form of vertical profiling by linking microprocessor-level metrics obtained from hardware performance counters to method-level phases in Java. This allows for analyzing Java applications at a finer granularity than the vertical profiling approach by Hauswirth *et al.* [10, 11], however, the granularity is still much more coarse-grained than the granularity that we can achieve using Javana.

The commercially available tool VTune [6] from Intel also allows for profiling Java applications. The VTune tool samples hardware performance counters to profile an application and to annotate source code with cache miss rate information. However, given the fact that VTune relies on sampling it is questionable whether this allows for fine-grained profiling information with little overhead and perturbation of the results.

All of these vertical profiling approaches rely on a microprocessor's performance counters. This limits the scope of these techniques to evaluating Java system performance on existing microprocessors. These approaches do not allow for building customized vertical Java program analysis tools as the Javana system does.

7. Summary

Understanding the behavior of Java application is non-trivial because of the tight entanglement of the application and the virtual machine at the lowest machine-code level. This paper proposed Javana, a system for quickly building Java program analysis tools. Javana is publicly available at `http://www.elis.ugent.be/javana/`. Javana runs a dynamic binary instrumentation tool underneath a virtual machine. The virtual machine communicates with the dynamic binary instrumentation tool using an event handling mechanism. This event handling mechanism enables the dynamic binary instrumentation layer to build a so called vertical map. A vertical map keeps track of correspondences between high-level language concepts such as objects, methods, threads, *etc.*, and low-level native instruction pointers and memory addresses. This vertical map provides the Javana end user with high-level information concerning every memory access the dynamic binary instrumentation tool intercepts. As a result, Javana is capable of tracking all memory references and all natively executed instructions and to provide high-level information for each of those.

Javana also comes with an easy-to-use Javana instrumentation language. The Javana language provides the Javana user with low-level and high-level information that enables the Javana user to quickly build powerful Java program analysis tools that crosscut the Java application, the VM and the native execution layer.

The first key property of Javana is that Java program analysis tools can be built very quickly. To demonstrate the real power of Javana we presented three example applications: memory address tracing, vertical cache simulation and object lifetime computation. For each of these applications, the core instrumentation specification was only a few lines of code.

The second key property of Javana is that the profiling results are guaranteed to be highly accurate (by construction) because the dynamic binary instrumentation layer tracks every single natively executed instruction. Current practice is typically one of manually instrumenting the virtual machine which is both time-consuming and error-prone. In addition, the accuracy of the profiling results might be questionable because it is hard to instrument a virtual machine in such a way that all memory accesses are tracked, as we have shown through our example applications.

8. Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments, as well as Hans Vandierendonck for integrating the SimpleScalar cache simulator into our Javana instrumentation specification. Jonas Maebe and Dries Buytaert are supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen). This research is also supported in part by Ghent University, the HiPEAC Network of Excellence and the European SARC project No. 27648.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set. Computer Architecture News, 1997. See also `http://www.simplescalar.com` for more information.

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

[4] B. De Bus, D. Chanet, B. De Sutter, L. Van Put, and K. De Bosschere. The design and implementation of FIT: A flexible instrumentation toolkit. In *Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 29–34, June 2004.

[5] M. Dmitriev. Selective profiling of Java applications using dynamic bytecode instrumentation. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 141–150, Mar. 2004.

[6] J. Donnell. *Java Performance Profiling using the VTune Performance Analyzer*. Intel, 2004.

[7] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, Applications and Systems (OOPSLA)*, pages 149–168, Oct. 2003.

[8] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, Applications and Systems (OOPSLA)*, pages 306–307, Oct. 2003.

[9] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 270–287, Oct. 2004.

[10] M. Hauswirth, A. Diwan, P. S. Sweeney, and M. C. Mozer. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 281–296, Oct. 2005.

[11] M. Hauswirth, P. S. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 251–269, Oct. 2004.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–355, June 2001.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[14] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, 1995.

[15] 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 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[16] J. Maebe and K. De Bosschere. Instrumenting self-modifying code. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG)*, pages 103–113, Sept. 2003.

[17] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Proceedings of the 2002 Workshop on Binary Translation (WBT) held in conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2002.

[18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[19] S. Reiss and M. Renieris. Languages for dynamic instrumentation. In *Proceedings of the Workshop on Dynamic Analysis (WODA)*, May 2003.

[20] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient Java. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 104–113, 2001.

[21] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: a structured view and opportunities for optimizations. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 194–205, June 2001.

[22] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, 2002.

[23] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, Mar. 1994.

[24] J. Suzuki and Y. Yamamoto. Extending UML with aspects: Aspect support in the design phase. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 299–300, 1999.

[25] P. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. H. d. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM)*, pages 57–72, May 2004.

[26] C. Zhang and H.-A. Jacobsen. TinyC^2: Towards building a dynamic weaving aspect language for C. In *Proceedings of the Foundations of Aspect-Oriented Languages Workshop at AOSD 2003*, Mar. 2003.