Dedicated Static Sampling Pointcut Designators

Amjad Nusayr
University of Houston Victoria
Victoria, TX, 77904, USA
Email: nusayra@uhv.edu

Abstract— Sampling-based instrumentation is often used to make runtime monitoring of applications more efficient. AOP has long been used for monitoring but does not have the ability to support code sampling in its current form. In this paper, we present an implementation of two new pointcut designators in a static weave form intended to be used as code sampling tools. The implementation is created as an extension using the aspect bench compiler (abc). These two new pointcut designators serve as additions to the family of joinpoints in AOP for the purpose of code sampling.

I. Introduction

Aspect Oriented Programming (AOP) is mainly concerned with the separation of cross-cutting concerns; the design of AOP was dedicated to separate the program main logic functionally from secondary yet; imperative code additions that otherwise make the code weak or susceptible to break [9]. Take for example the code needed for a student to register in a class. The basic logic of this code is simple; first make sure that the class still has seating space for the student. If so, add the student to class roster. Now creating the code for the above logic is straight-forward, but does not represent reality. In real life, there should be a segment in the code that always checks to authenticate the user during the session. There should be as well code that logs any transactions to some database. These two actions are not part of the basic program logic but should be incorporated to validate the program and the actions of the user.

Another major example is code based exceptions. The mere semantics of an exception is to catch some behavior that is not covered and does not meet the basic program logic. The simple example of “DivideByZero” exception shows the importance of having such cross-cutting concerns implemented in code.

The concept of cross-cutting concerns in AOP has been also utilized in other arenas. The research shows much usage of AOP in domain specific languages [6, 7]. This usage spanned from expressing constraints on the behavior and structure of a program all the way to parallel dynamic analysis of multicores [2]. The growth of AOP has itself been extended to include more sophisticated tools in aiding code development. This paper describes two new sampling designators, their main functionality, their implementation, and how they are used. It also presents a performance analysis compared to other designators.

DOI reference number: 10.18293/SEKE2017-175

II. Related and Background work

Since AOP was established, the notion to extend the set of pointcut designators was defined [3, 13]. AOP introduces a modular approach in helping to separate crosscutting concerns by an *Aspect*. An aspect relies on the concept of *weaving*; the instrumentation of crosscutting code into the main program code logic. A typical aspect module contains several fundamental components; the first component is the *pointcut designator*. Pointcut designators are features in the program execution where the advice of an aspect can be weaved in. A composition language allows a pointcut expression to combine and constrain these to define a pointcut (execution occurrences of the program features) that satisfies the expression and where the advice will be executed[1]. The *advice* represents the code to be weaved at locations defined by the pointcut expression.

The code below shows an example of an *after* advice that is weaved after every time the method openSocket is executed. The advice will print out the reflective information about, where in the code the particular execution has occurred. Note that in this particular example, the advice will apply on any method that begins with “openSocket” regardless of the return type and the number of parameters.

```
after() execution(* openSocket*(..)) {
    //Advice to be woven
    System.out.println("socket open at"+
    thisJoinPoint.getSourceLocation());
}
```

The amount of AOP integration within a system depends on the way it is implemented. It could simplify the base code and the semantics of the program or it could do the opposite by introducing too much coupling among different aspects. The latter defeats the purpose of AOP. A well-known study examines the interactions between aspects and methods and identifies classes of interactions that enable modular reasoning about the crosscut program [12].

AOP has been implemented to be used in several languages. One of the most used is AspectJ; an implementation of AOP for the Java language [8]. An alternative to AspectJ is the aspect bench compiler (abc) [3]. The main difference between AspectJ and abc is that abs is designed for extensibility and modification by creating extensions. The extension of abc includes introducing new grammar to the language, followed by introducing new rules for the semantics and code needed for abstract syntax trees. Adding advice includes defining how code

1. AOP literature uses the term joinpoint to refer to a point in the code execution. We feel that the term “pointcut designator” has a more abstract meaning and usage.

should be generated to invoke that advice and where in the joinpoint shadow this code should go. The Shadow pointcuts pick out a specific joinpoint shadow within a method body (i.e. enables to shadow every executable statement in the code).

Sampling could be defined as a general utility of summarization for a broad spectrum of analytical tasks [4]. In the world of computer science, sampling research is usually conceived to target data sampling techniques. An example is sampling big data and graph, and network sampling. BlinkDB,is a well-known approximate query engine on large quantities of data used for sampling [1, 4].

The use of the term *code sampling* is generally found in compiler research [5]. Code sampling is used to determine many aspects of the code such as unreachable code fragments (with the help of code coverage), dead-code elimination, and the replacement of frequent accessed code segments with other faster segments.

III. What to sample

The first set of questions the think of are, A) which locations or points in a program should be selected for sampling? B) what exactly should be sampled? And C) how often should it be done? These three questions were at the basic design of the new pointcut designators. The following is a discussion for each question:

A. *Which locations or points in a program should be selected?*

AOP defines a joinpoint as a point in the execution of the program. The abstract norm of the word "point" refers to a location in code. That location could be a decision structure or a loop back-edge or any executable statement. In previous work, we defined the point to be three possibilities [10]: *Code*: In this case, the weaving is based on a particular point or set of points that exist in code. *Time*: Using time as a point means that sampling is controlled by some kind of timing theme. This could be relative time or wall-clock time. *Data*: Using data as the definition of a point means that sampling would be over some data space and weaving runs when this data are accessed and the sampling criteria is met. In this paper, the focus is on code space sampling since sampling is done in a static manner (at compile time).

B. *What exactly should be sampled?*

The sampling process in general can occur in one of two ways; the first is *static sampling*. This means that the selection of the actual points (locations) in code for sampling is going to happen at compile level. This could so abstract and general; for example, sampling for every nth executable statement in code. It could be more detailed by picking every 30% of switch statements in the code. The second type of sampling is *dynamic sampling*, which will weave code that matches some point of execution at runtime (while the program is running). That is the weaving itself happens at runtime. There is no predetermined location in the code to be examined. Dynamic sampling will incur some overhead. This overhead is due to the selection point and the weaving process. The focus in this paper is only on static weaving as dynamic weaving was discussed in other work.

C. *How often should it be done?*

Since static weaving is the choice, one has to be careful on the frequency on selecting the weaving points in code. The decision whether or not to instrument a particular point in the program is made at program compile time.

IV. Static weaving

In static sampling the decision whether or not to instrument a particular point in the program is made at program generation time. In AOP this translates to weaving-time decisions about whether a joinpoint shadow should match the pointcut expression or not. Our sampling pointcuts essentially take the set of joinpoint shadows that would match the non-sampled pointcut expression and reduce it by the selected mechanism. In this section we introduce two new static weaving pointcut designators; weaveProbability(p) and sliceOf(N,M).

A. *The weaveProbability Pointcut Designator*

The first new static pointcut designator is

weaveProbability(P)

This designator controls whether weaving at a statically matching joinpoint shadow occurs. With probability P the advice will be woven, while with probability 1-P it will be skipped and not woven. This will then correspond to a sampling of the joinpoints at runtime, but where all joinpoints of a given shadow will be executed or not, depending on the compile-time probabilistic decision. Since this is a compile-time decision, the probability used does not automatically correspond to an equal expected runtime reduction. Only if the execution of the joinpoint shadows is relatively uniform will the static probability actually result in a similar dynamic probability. If a small number of joinpoint shadows incur most of the advice execution for a particular pointcut expression, then the runtime overhead will vary greatly, depending on how many of those high-execution shadows have been probabilistically selected.

This pointcut designator is simple and efficient, and is useful for reducing monitoring cost where there is a reasonably large number of static joinpoint shadows and the expected behavior over them is mostly uniform. If there are only a few shadows (e.g., say a program has three sites that match call(Table.insert)), then this sampling pointcut designator would probably not be useful

B. *The sliceOf Pointcut Designator*

The more involved, and perhaps more useful, static pointcut designator is

sliceOf(N,M)

This designator, rather than being probabilistic and thus uncontrollable, is deterministic in its sampling behavior. This designator selects the M_{th} fraction of joinpoint shadows, of size $\frac{1}{N}|JPSet|$ In other words, of the total number of shadows that match the pointcut expression, this selects 1/N of them, and the slice that it selects is the M_{th} one. For example, if 100 joinpoint shadows match a pointcut, then sliceOf(10,1) will select the first 10, sliceOf(10,2) will select the second 10, and sliceOf(10,10) will select the last 10. Fig. 1 shows the sampling of the first 10%

of shadow matches that could be used to study the behavior of program startup

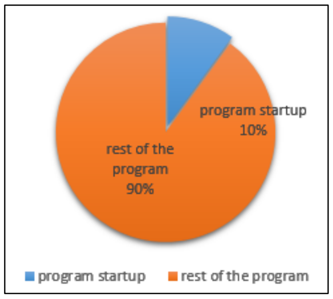

Figure 1: sliceOf(10,1) of a 100 shadow matches applied with the basicblock designator

This pointcut designator allows control over how the distribution of sampling will occur. It enables the creation of N versions of the application, over which it is guaranteed that all shadows that match the pointcut expression are covered, and thus all joinpoints that occur in execution are also covered. These versions can then be deployed and used, and the data collected be used in aggregate to understand some aspect of the program in its entirety.

This designator requires knowledge of how many joinpoints match the pointcut expression in order to slice properly. In the abc framework this is not known until the end of compilation, and so this pointcut designator requires a 2-phase implementation. When the program and aspect is first compiled, it will generate a data file that contains the count of joinpoint shadows. On a re-compile it then uses the data file to perform the slicing and weave the correct shadow sample.

This type of instrumentation is typical of schemes devised for end-user remote application monitoring, where there are competing desires to monitor the whole program but also to not affect the performance for any one user [11]. By distributing different versions of the instrumented program to different users, and collecting and aggregating the obtained data, information about the whole program can be obtained. This scheme also serves to provide some level of privacy assurance to users, since for any particular user only a small portion of application behavior might be recorded.

V. Baseline Measurements and Evaluation

The program Image2Html was used as a small example to first demonstrate sampling and to obtain baseline performance measurements. This program converts a JPEG image into textual HTML-enhanced "ASCII art".

This test used the *basicblock* designator [10] combined with one of the sampling designators; e.g., the pointcut expression is like "before: basicblock() && weaveProbability(.3)". The advice is a simple basic block profiler that counts how many times each block has executed

Fig. 2 shows these results from Image2Html The horizontal lines are baselines of performance of the program with full instrumentation (no sampling) and with no instrumentation. In the fully instrumented version, advice gets executed 3,588,277 times over 411 basic blocks in the program. The three linearly increasing lines on the graph are three dynamic sampling designators which are not in the scope of this paper.

The static sampling shows good and expected performance. The Figure shows datapoints for weaveProbability of 0.15, 0.25, and 0.6; and shows data points for sliceOf for N = 2, N = 4, and N = 8. Since runtime costs for static weaving can change when the weaving is re-applied, each data point shows an average time plus error bars indicating the minimum and maximum observed run times. For weaveProbability, the configuration was compiled and ran four times, and for sliceOf the configuration was compiled and ran for each possible M value (e.g., for N = 8, with 8 tests, with M = 1, 2, 3, ...8). The N = 8 data point is at the 12.5% sampling position on the X axis, while the N = 2 data point is at the 50% sampling position.

Because sampling decisions are made statically, with no runtime check, their performances are in general better than dynamic sampling. As predicted, however, static sampling can suffer from high variation in runtime costs. In this program, weaveProbability does not incur too high of a variation, but sliceOf does. One configuration in N = 8 incurs almost half of the full instrumentation cost, while other incur almost no cost. The range of runtime costs for N = 4 span almost the full range between the no instrumentation and full instrumentation baselines. Finally, the two runs for N = 2 (one being the minimum and one being the maximum) are also quite far apart in runtime cost.

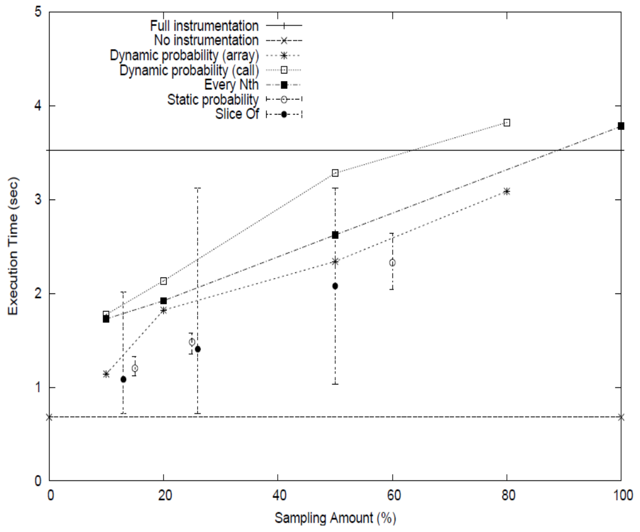

Figure 2: Basic block sampling performance for Image2Html. Error bars on the static pointcut designators (static probability and slice of) show the minimum and maximum of the averaged values

VI. Implementation Issues

The AspectJ documentation separates the fundamental pointcut designators into three categories: kinded, scoping and context [8]. Their documented definitions are: Kinded designators are those which select a particular kind of join point; for example: execution, get, set, call, handler. Scoping designators are those which select a group of join points of interest (of probably many kinds); for example: within, withincode. Contextual designators are those that match (and optionally bind) based on context; for example: this, target, @annotation.

Each of the three categories above can select a set of joinpoints to execute the advice on, although in practice pointcut expressions usually have at least one kinded pointcut designator. Our sampling pointcut designators are different in that they are meant to operate on an already selected set of joinpoints. The static sampling designators are expected to effect the size of this set, but only in a negative manner: they reduce the given set by some amount.

Typically, new pointcut designators created using abc would perform their own "shadow matching" where they selected some execution points (joinpoint shadows) to include in the joinpoint set (thus including all joinpoints which occur at those shadows). The internal pointcut expression evaluator would then combine with the shadows from other parts of the expression to compute the actual set of joinpoint shadows for the given pointcut expression.

For our sampling extensions, we did not add any shadow matching in the extension, but rather we hook into the optimization phase of pointcut expression evaluation, and it is in this phase that we apply the desired sampling effect. This is achieved by the removal of some of the joinpoint shadows from the selected set.

VII. FUTURE WORK

The basic sampling mechanisms we presented in this paper can be extended in a variety of directions. For the static sampling pointcut designator sliceOf, a better mechanism to control expected runtime costs is needed. One way to do this is with profiling information delivered to the compilation phase. A fully instrumented version could be generated, and under test conditions an expected profile of per-joinpoint-shadow execution costs could be obtained. This would then be used to allocate the expected higher cost shadows among different slices, which should help alleviate the high variation in the performance of the different slices.

Another useful extension would be that of linked sampling, where for example, if probabilistic sampling chose to execute the advice on a top-level method, then all advice (instrumentation) on code that is invoked from that method should be executed as well. For example, if the top-level method invocation represents the handling of one transaction in a TPS, then we could implement sampling on transaction, where all instrumentation would execute for a selected transaction, and not for others.

VIII. CONCLUSION

We presented new pointcut designators for AspectJ with the use of the aspect bench compiler that allow the creation of sampling-based instrumentation, a common need in program monitoring scenarios. With sampling, AOP can be used for efficient monitoring instrumentation even on heavily used code that would otherwise be prohibitive to monitor. Our research goal for this line of work is to enable the high-level formalisms of AOP to be useable for the wide variety of low-level sampling needs. As the future work discussion shows, there is still work left to be done in this venue, but the sampling pointcut designators described in this paper are a significant contribution in this direction.

It is likely that the sampling designators could be useful for other application and system needs. One example would be to sample transaction information of an ecommerce to obtain a real-time statistical profile of user requests. It is likely that developers could think up wide and novel uses for AOP with sampling.

REFERENCES

[1] Albattah, W. 2016. The Role of Sampling in Big Data Analysis. Proceedings of the International Conference on Big Data and Advanced Wireless Technologies (New York, NY, USA, 2016), 28:1--28:5.

[2] Ansaloni, D., Binder, W., Villazón, A. and Moret, P. 2010. Parallel Dynamic Analysis on Multicores with Aspect-Oriented Programming. 2010 Conference on Aspect-Oriented Software Development (AOSD) (2010), 1–12.

[3] Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. and Tibble, J. 2005. Abc: An Extensible AspectJ Compiler. Proceedings of the 4th International Conference on Aspect-oriented Software Development (New York, NY, USA, 2005), 87–98.

[4] Cormode, G. and Duffield, N. 2014. Sampling for Big Data: A Tutorial. Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (New York, NY, USA, 2014), 1975.

[5] Debray, S.K., Evans, W., Muth, R. and De Sutter, B. 2000. Compiler Techniques for Code Compaction. ACM Trans. Program. Lang. Syst. 22, 2 (Mar. 2000), 378–415.

[6] Hadas, A. and Lorenz, D.H. 2015. First-class Domain Specific Aspect Languages. Companion Proceedings of the 14th International Conference on Modularity (New York, NY, USA, 2015), 29–30.

[7] Hadas, A. and Lorenz, D.H. 2016. Toward Disposable Domain-specific Aspect Languages. Companion Proceedings of the 15th International Conference on Modularity (New York, NY, USA, 2016), 83–85.

[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. 2001. An Overview of AspectJ. Proceedings of the 15th European Conference on Object-Oriented Programming (London, UK, UK, 2001), 327–353.

[9] Mens, K., Lopes, C.V., Tekinerdogan, B. and Kiczales, G. 1998. Aspect-Oriented Programming Workshop Report. Proceedings of the Workshops on Object-Oriented Technology (London, UK, 1998), 483–496.

[10] Nusayr, A. and Cook, J. 2009. Using AOP for detailed runtime monitoring instrumentation. Proceedings of the 7th International Workshop on Dynamic Analysis -- WODA'09. (2009), 8–14.

[11] Orso, A., Liang, D., Harrold, M.J. and Lipton, R. 2002. Gamma System: Continuous Evolution of Software After Deployment. Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (New York, NY, USA, 2002), 65–69.

[12] Rinard, M., Salcianu, A. and Bugrara, S. 2004. A Classification System and Analysis for Aspect-oriented Programs. SIGSOFT Softw. Eng. Notes. 29, 6 (Oct. 2004), 147–158.

[13] Ubayashi, N. 2004. An AOP Implementation Framework for Extending Join Point Models. Proceedings of ECOOP 2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04 (2004), 71–81.