Using AOP for Detailed Runtime Monitoring Instrumentation

Amjad Nusayr
New Mexico State University
anusayr@cs.nmsu.edu

Jonathan Cook
New Mexico State University
joncook@nmsu.edu

ABSTRACT

Although AOP has long been used for monitoring purposes, the level of detail that virtually all AOP frameworks support is not enough to support broad classes of monitoring applications. Often the method or function call is the basic unit of code weaving, and if some data access weaving is supported it is usually very limited, e.g., to object field access. In this paper we demonstrate the need for AOP to be extended if it is to support broad runtime monitoring needs, and then present two new joinpoint types for AspectJ, the *basic block* and *loop back edge* types. These allow much more fine-grained weaving of advice, which in turn supports a much larger category of runtime monitoring applications. We demonstrate the feasibility of these extensions with example prototype implementations using the AspectBench Compiler (abc).

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Measurement, Languages

Keywords

Runtime Monitoring, Aspect Oriented Programming

1. INTRODUCTION

Instrumentation for runtime monitoring elegantly fits into the idea of advice in aspect oriented programming (AOP), which is program behavior that is orthogonal to the underlying program code base. From the beginning of AOP it has been observed that runtime monitoring is a natural domain for AOP, with the logging monitor being the "Hello World" example for AOP. Instrumentation for runtime monitoring perfectly fits the idea of a crosscutting concern, and given the difficulty of "rolling your own" instrumentation one might expect AOP to have taken over the task of instrumentation for runtime monitoring. Despite this natural fit and several examples of such use ([5, 7, 10, 14, 20]), runtime monitoring has not seen a massive shift towards AOP systems. One could pessimistically say that this is because AOP is not standard in the industrial development toolkit.

Rather, we believe this stall is because of two reasons. One, existing popular AOP frameworks have not offered a great enough level of weaving *detail* to be more generally useful for runtime monitoring (RM). Two, existing AOP ideas and frameworks have limited the *dimension* of weaving to just the source code, excluding many monitoring needs that might not be easily translatable to code-centric instrumentation (e.g., sampling-based profiling may need weaving based on execution time intervals rather than places in the code). We are not the first to note this; similar limitations were noted earlier in [20].

This paper summarizes the idea of multiple weaving dimensions (previously outlined in [17]), and then presents two new pointcut designators [1] that extend the level of detail that weaving can be performed on. These two are the *basic block* and the *loop back edge* pointcut designators, which allow the weaving of advice (RM instrumentation) at a greater level of detail than before. The prototypes are implemented using the *AspectBench Compiler*, or *abc* [2].

2. BACKGROUND

2.1 Runtime Monitoring

Runtime monitoring requires some sort of *instrumentation*, which probes the executing system and reports back something about the data or execution of that system. Instrumentation is often inserted *in-line* into a program, so that each time the program hits a certain point of execution, the instrumentation is executed. Sometimes, such as with debugger breakpoints, the instrumentation is hardware assisted to make it much more efficient. Some monitoring approaches may use timer-based instrumentation, such as sampling-based code profiling (e.g., [16]), while others may

This work was supported by the National Science Foundation under grant CCF-0541075. Views expressed herein are those of the authors and do not necessarily represent those of the Foundation.

WODA '09, July 20, 2009, Chicago, Illinois, USA.

[1] AOP literature inconsistently uses the terms joinpoint types, primitive pointcuts, and pointcut designators to refer to the fundamental kinds of joinpoints that advice can be woven on, which manifest themselves as the terms in the pointcut expression language; we use the latter.

use memory page protections to signal an event (e.g., [21]), and still others use probabilistic or counter-based sampling (e.g., [1, 15]). Effectively creating the necessary instrumentation, and making it efficient, are complex and technologically difficult tasks, often involving creating specialized instrumentation tools by hand.

Instrumentation costs span the gamut from barely noticeable to extremely painful in terms of their impact on application performance. Sampling-based code profilers are extremely efficient, gathering very simple data periodically while the program executes, and performing offline analysis to construct the execution profile [16]. Debugger expression watchpoints, on the other hand, can cause 4-6 orders of magnitude slowdown if they have no hardware support and thus need to evaluate an expression after each instruction [18]. Complex analyses such as the invariant inference of Daikon [9] require heavily instrumented programs to collect data variable history, and also use offline analyses to run the inference algorithms.

2.2 Aspect Oriented Programming

Aspect Oriented Programming frameworks also *instrument* an underlying base program, but in AOP this purpose is more generic, to weave in any crosscutting functionality that should be factored out of the base program and not be replicated in the many locations in the program source where it is needed. A basic AOP model defines some specific fundamental *pointcut designators* (PCD), which 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*, which is a set of program *joinpoints* (execution occurrences of the program features) that satisfy the expression, and where the advice will be executed.

In existing AOP frameworks, the fundamental pointcut designators are chosen somewhat pragmatically: they must be actually useful to an aspect programmer, but they must also be relatively practical to implement in the AOP system. Thus, in existing AOP systems, pointcut designators are typically points in the program where inserting instrumentation is "not too hard"; for example, method calls are very often used as one of the fundamental pointcut designators.

The most popular AOP system, AspectJ, implements AOP for Java programs. Its pointcut designators include method calls, method executions, object field accesses, exceptions, and a few others [13].

3. NEW DIMENSIONS FOR AOP

There is a clear synergy between the ideas of AOP and the needs of runtime monitoring, and the only barrier to allowing AOP to be the unifying framework for almost all of runtime monitoring is the fact that current AOP frameworks are far too limited in the level of detail they support weaving at, and the fundamental style of weaving they support. In [17] we introduced the idea of extending AOP into heretofore overlooked dimensions in order to support runtime monitoring. In this section we summarize those ideas.

The general view of AOP thus far has been one where advice is woven into a base program based on joinpoint types that are *code based* or *data based*, e.g., method call and field access. Even the data-access pointcut designators often get mapped to code-based concepts (i.e., all expressions that access an object field).

We propose a more extensive *multi-dimensioned* view of AOP. Aspect weaving should not be limited to only be done based on code features, and this is especially important for AOP to support the broad needs of runtime monitoring. We identify four dimensions of weaving that are needed.

- **Code:** The traditional weaving over code features.
- **Data:** Weaving over concepts in data space.
- **Time:** Weaving based on time constraints.
- **Sampling:** Weaving that supports sampling-based instrumentation.

The *code* dimension weaving is well understood already, and most AOP frameworks support a variety of code-based pointcut designators. We only mention here that, for monitoring purposes, existing AOP frameworks are still quite limited in the features that they support, and this paper contributes two new code-based pointcut designators to the AspectJ framework. Statement level coverage analysis and many other analyses depend on being able to instrument down to the statement (or basic block) level, yet current AOP systems do not support this.

The *data* dimension weaving has some, but very limited, support in existing AOP frameworks. For example, AspectJ has pointcut designators for object field accesses, but not for accesses to local variables, array elements, or arguments. Above the basic concepts of individual variables, the data dimension could support weaving on higher level data-oriented concepts, such as when a node in a data structure is accessed, when references to objects have changed, or how much space an application has allocated.

The *time* dimension will weave aspect code not based on location in code but on timers, either relative or absolute. An obvious example of the utility of this dimension is profiling, where a timer-based interruption of the program samples where the program is at that point in time, and constructs a statistical profile of the execution behavior of the program. Other time-based uses would be to periodically check data structure health or application progress. This might be done over relative time, such as for every 10 minutes of program execution time, or it might be over absolute time, such as at 1:00am every Sunday night while the system is more likely to be idle.

The *sampling* dimension controls whether or not the advice is actually executed at a joinpoint, or not. Current AOP assumes that every time a joinpoint satisfying the pointcut expression is reached, the advice will execute. However, research in runtime monitoring has shown the utility of *sampling* based approaches, where instrumentation is executed probabilistically, either randomly or (more efficiently) with a counter-based approach. This dimension could also be understood statically, in the sense of statically selecting a sample of matching instrumentation points. This would support distributed remote monitoring, where individual users incur only a fraction of the cost of full instrumentation, but full coverage of all instrumentation points is achieved over the all users.

Discussion

As with existing AOP pointcut designators, not all compositions of new PCD's in these dimensions would make sense,

and the semantics of some compositions need to be carefully specified. For example, composing the time pointcut designators with others, such as method invocations, brings in some design decisions because it is virtually guaranteed that the two pointcut designators, each designating an "instantaneous" event, would never be simultaneously satisfied (one solution would be to make the time domain designator mean "at least this much time" when composed with other designators). New sampling PCD's would not be useful by themselves since they do not select any inherent joinpoints, but rather provide constraints over joinpoints selected by other PCD's.

AOP weaving over non-code dimensions can be a unifying theory for some current practices such as garbage collection and other services, where e.g., the triggering of a garbage collection is done on a time interval (time dimension weaving) or a space usage trigger (data dimension weaving). Imagining system services as aspects was noted in [6], though the mapping to weaving dimensions was not.

Some may argue that AOP should stick to being language-centric, and to composing applications only over language features, but there is no fundamental reason why extra-language features of the program runtime should not be considered for AOP. The line between a language, its canonical supporting libraries, and its runtime environment are already blurred—e.g., the Java language specification requires that garbage collection be a runtime service, that threads behave in specific ways, and that written string constants be objects of the String class.

4. NEW POINTCUT DESIGNATORS

Our first extension to existing AOP ideas adds two fundamental pointcut designators that extend the level of detail for where advice can be woven into the program code. Even in the code dimension, no current AOP framework has come close to attempting to support all, or even most, of the constructs in any given programming language. Typically, the "easy to instrument" features are supported: function and method calls and returns, some level of variable access if it is easily supported, and other language features such as exceptions, if they exist.

In general, exposing all program constructs for weaving would ensure that any imagined monitoring in the code dimension could be supported. However, this may not be possible without building a new AOP framework from scratch;[2] we are not pursuing that direction now, and so we introduce only two new pointcut designators here.

The first designator we implemented is a *basic block* pointcut designator (PCD), that matches basic blocks in the code. This will support, for example, a code coverage analyzer that must monitor the execution of basic blocks during a program run. The second is a *loop back edge* pointcut designator, which matches the jump back up to the top of a loop construct. Many monitors, such as a code-based profiler, need to instrument every possible cyclical path in order to ensure that the instrumentation is never "locked out" of the execution for an indefinite period of time. The only possible cyclical paths are iterative loops and recursive function calls. Current AOP frameworks can instrument method/function calls, but do not support loop instrumentation; our extension adds this capability.

[2]When we encountered errors in trying to implement a statement-level pointcut designator in one framework, the developers told us that it "was never intended to support that level of detail."

4.1 Implementing The Extensions

We used the AspectBench Compiler system (*abc*), since it was the only one we found that was purposely created to support extending the basic aspect model [2]. *Abc* is a rather large system, attempting to essentially offer the same base functionality as AspectJ, and in addition allowing for extension development. In this paper, when we refer to AspectJ we mean the implementation of AspectJ by the *abc* system. Creating an new pointcut designator involves extending the pointcut grammar, implementing the shadow matching process that marks potential join points for the new pointcut designator, performing the actual weaving, and implementing reflective information.

Typical AOP systems support three types of weaving: before, after, and around; the first two execute the advice before or after the matching joinpoint, respectively, while the around weaving allows the advice to control whether or not the program construct making up the joinpoint is actually executed or not. Since we are currently targeting only monitoring instrumentation, which should not affect the execution of the program, we do not support the around type of weaving with our new pointcut designators.

4.2 A Basic Block Pointcut Designator

Our most basic instrumentation addition is a pointcut designator for basic blocks. We name this designator `basicblock`. A source code example showing the usage of the block pointcut is below.

```
aspect TraceBlocks {
 before(int id) : basicblock() && args(id)
 {
  System.err.println("Entering block " + id +
    " at " + thisJoinPoint.getSourceLocation());
 }

 after(int id) : basicblock() && args(id)
 {
  System.err.println("Exiting block " + id);
 }
}
```

For a woven program, the above code will print out the event of entering the block and exiting the block, with reflective information printing the line number in the source code of the start of the block. The *args* pointcut designator is generally used to match the arguments available at a particular join point (a call or execution pointcut that has arguments). For the basic block designator we use this to pass a unique identification number of each basic block within a class method. The identifier is only unique within a method, and blocks within a method are guaranteed to be numbered sequentially starting at 0.

Additionally, our basic block extension can, during weaving, record into a file the number of basic blocks in each method that it instruments. With this static information, our extension can support coverage analysis and other similar types of monitoring that require both the static information about basic blocks and the dynamic execution monitoring of the blocks. We expect in the future to support the recording of more static information gathered during compilation; for example, it would be easy to record enough

information to reconstruct the control flow graph for later dynamic analyses.

For the *before* weaving with the basicblock PCD, we had to make sure all jumps to this basic block were re-routed to its new beginning (the advice). *Abc* makes this type of program transformation easy, and this was not a big problem to solve. For *after* weaving, we had to check the form of the last instruction in the block. If it is a return, an unconditional jump, or a conditional jump, then we must place the after advice just above this last instruction, or else it will not be executed (except for the fall-through case on a conditional jump). For a conditional jump, Java bytecode encodes the relational comparison directly into the conditional jump instruction; thus we cannot weave *after* the comparison but before the jump. We can only weave after any arithmetic expression evaluation for the condition, but not after the final relational comparison that determines whether the jump happens or not[3].

4.3 A Loop Back Edge Pointcut Designator

Our loop back edge designator, `loopbackedge`, matches the jump back up to the loop condition; this has been shown to be useful in several dynamic analyses (e.g., [1, 15]). For this PCD, we currently support only *before* weaving, since the weaving after would be harder (it would basically mean weaving at the top of the loop body) and there are no actual executable statements in between the before and after locations anyway.

A source code example of the loop back edge pointcut is shown below. It supports the same argument as the basic block pointcut designator, which is the basic block ID number of the last block in the loop; this can be used to uniquely identify loops.

```
aspect TraceLoops {
 before(int id) : loopbackedge() && args(id)
 {
  System.err.println("In loop " + id +
   " at " + thisJoinPoint.getSourceLocation());
 }
}
```

In the *abc* framework, weaving is done on an intermediate representation similar to Java bytecode, where all loops have already been converted to conditional jumps and goto's. Thus, our extension does not need to distinguish between the various source-code flavors of loops; all loops are handled automatically by detecting all backwards jumps. The weaving implementation finds all conditional or unconditional jumps that jump to a previous (or the same) basic block, and weaves the advice just before the jump instruction.

4.4 Summary

Our two new pointcut designators, *basic block* and *loop back edge*, certainly do not cover the full detail in the code dimension that all of runtime monitoring might need in instrumentation capability; yet they do extend weaving to a much greater detail than previously available, and make many more monitoring needs amenable to using AOP rather than performing difficult instrumentation by hand. It is these types of extensions to AOP that are needed to make it truly useful to broad runtime monitoring needs. We have only scratched the surface with these two PCD's, and even with them one can imagine a much richer argument set; for example, making the context of a basic block available in a pointcut expression (e.g., a block that is in a loop or not, a block that is part of a conditional expression, or a block with a method call) would allow this simple PCD to be used in a large variety of monitoring scenarios.

[3]We do see some value in a PCD that instruments condition expressions in a program, and that has access to the boolean result; this will be possible but will incur some bytecode duplication to mirror the condition evaluation for the advice.

5. EXAMPLES

In this section we demonstrate the feasibility of our extensions for a variety of runtime monitoring needs, and over a variety of Java applications. The applications we chose are the following:

- *JTetris*, an implementation of the popular Tetris game; this application is centered on user interaction and is not compute-bound;
- *Image2Html*, a program that takes a JPEG image and translates it into a colored HTML "ascii art" picture; it has a GUI mode but we only run it in its command-line mode; and
- *Java Linpack*, an implementation in Java of the FORTRAN Linpack routines (matrix and vector manipulation); this program is solely a command-line, compute-bound program.

These applications give us a broad enough base across different application performance types to demonstrate our techniques on. All experiments were run on a Dell rack server, with a quad-core Intel Xeon 64-bit CPU, 2.13GHz, 4GB RAM, running OpenSuse Linux 10.3, kernel version 2.6.22.18. All execution time values are the result of using the Unix *time* command, averaging the execution of five equivalent runs, and subtracting out an average value for executing an empty Java program, in order to factor out the JVM startup cost.

5.1 Coverage Analysis

Our first example is the straightforward example of coverage analysis, based directly on our basic block aspect mechanism. Using the basic block pointcut designator, we developed an aspect program that weaves a before advice for each basic block executed. The advice in the aspect records execution counts for each basic block that is executed. The static count of basic blocks for each method (produced during the compilation/weaving step) allows coverage statistics to be produced from the dynamic execution counts, along with count-based profile information.

Table 1 shows the results of the coverage analysis, while the baseline plots in Figure 1 show just the execution times for no instrumentation and for full. We did two different runs, one with full program instrumentation (all classes), and one where just one "key class" is instrumented. For each application we chose one class that is central to the program and would likely be the focus of intense scrutiny if one was evaluating the program. Note that for *Java Linpack*, the application consists only of one class, so the results are equivalent for the two runs.

The compute-bound *Java Linpack* shows the worst overhead, somewhat more than an order of magnitude for full

Table 1: Coverage Example Results.

Java Application	Full app # blocks	Key class # blocks	Time, no instr	Time, full instr	Time, key class instr	Coverage % full instr	Coverage % key class
Java Linpack	156	156	0.0675	0.879	0.879	69.23	69.23
JTetris	240	58	0.3275	0.649	0.459	65.48	72.41
Image2Html	409	267	0.6611	3.655	2.797	51.83	64.79

Table 2: Probability, Counter, and Timer Sampling Results.

Java Application	# methods + loops	Instrumented Execution Times (sec)							
		Prob = 0.50		Prob = 0.05		Count = 1 in 2		Count = 1 in 20	
		block	loop	block	loop	block	loop	block	loop
Java Linpack	38	0.572	0.335	0.271	0.187	0.519	0.307	0.227	0.159
JTetris	84	0.547	0.435	0.439	0.399	0.547	0.435	0.443	0.379
Image2Html	39	2.311	0.819	0.967	0.735	2.183	0.735	0.895	0.719

Java Application	Time = 1usec		Time = 10usec	
	block	loop	block	loop
Java Linpack	28.616	12.739	28.184	12.707
JTetris	0.887	0.635	0.827	0.595
Image2Html	111.785	9.067	111.741	8.955

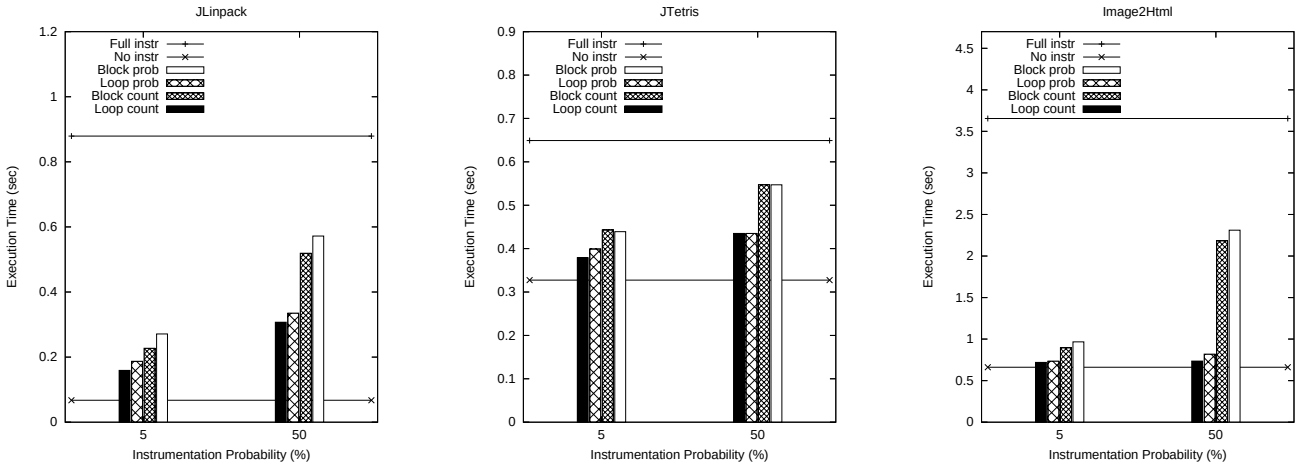

Figure 1: Execution Time Barcharts. The baselines 'Full' and 'None' are from Table 1, while the bars are from Table 2.

instrumentation; this quite acceptable for full basic-block instrumentation on an application that has heavily executed blocks with only computational instructions in them. The others show better overhead, with *JTetris* only twice as slow with full instrumentation. Our qualitative evaluation here is that in running even the fully instrumented *JTetris*, the overhead is not obvious to the user; the game still runs smoothly from the user experience point of view.

5.2 Profiling

In this example monitor we demonstrate how new dimensions of AOP weaving could be used to accomplish sampling-based statistical profiling. Since we have not yet created new pointcut designators for the sampling and time dimensions, we simply coded the counter, probability, or timer check by hand into the advice woven by a code-based pointcut expression; true designators would be cleanly orthogonal from the instrumentation body, could likely be made much more efficient (especially for the time dimension), and could potentially be optimized by the weaving compiler.

Table 2 shows results from probabilistic, counter, and timer based sampling methods, while Figure 1 shows plots of just the probabilistic and counter instrumentation relative to none and full instrumentation (though the full instrumentation from Table 1 is slightly different). We used two instrumentation methods, either instrumenting all basic blocks, or instrumenting only method executions and loop bodies. The second provides slightly less granularity for profiling, but is much more efficient. As is seen in the table, the number of joinpoints for basic block instrumentation is three to ten times higher than for the method plus loop instrumentation. (For block instrumentation, the number of blocks instrumented is the same as from Table 1, as are the uninstrumented execution times.) We used two levels of sampling for each method, making them an order of magnitude in difference; for the probability and counter methods, we picked exactly 5% (1 in 20) and 50% (1 in 2), while for the timer method we picked numbers that fire a reasonable number of advice executions (1 and 10 microseconds).

For the probability and counting methods, performance is very good, with the method plus loop instrumentation having only about a 5x cost for *Java Linpack*, and only 30-40% for the others; basic block instrumentation is about the same cost as for the coverage analysis, which is to be expected. The instrumentation costs do not drop very much for the 10x sampling rate drop because we are still entering the advice each time, to check the count or probability and decide to execute the advice or not; this is where a true native pointcut designator can potentially offer great improvement in overhead.

For timer-based sampling, the first thing noticeable from this table is that the monitoring overhead is huge, approaching three orders of magnitude for the compute-bound applications. The numbers between the different frequencies of advice execution are virtually identical; thus this overhead is due not to advice execution. Instead it is simply due to the act of getting the current application time and checking it against the interval, which causes a system call each time and happens at every joinpoint (basic block or method/loop). This result points up the necessity for having inherent native support for time-dimension aspects in an AOP framework. Trying to approximate it ensues too much overhead, unless one has a fast in-process method for obtaining time usage.

6. RELATED WORK

There is much recent activity and novel ideas for extending AOP in manners similar to our work here, but we do not find previous work that specifically laid out the ideas of various weaving dimensions and types of detailed code weaving that are particularly useful for runtime monitoring. Rajan [19] and then Dyer and Rajan [8] are investigating very similar ideas, explicitly working on arguing for more extensive join point models (thus allowing more pointcut designators) and embodying those in an intermediate language and virtual machine support for weaving. Rajan and Sullivan were, as far as we can tell, the first to make a clear note that current AOP models are insufficient to support many monitoring tasks such as coverage and profiling [20].

Harbulot and Gurd [11] used *abc* to extend AspectJ with a pointcut designator for loops, specifically focusing on numerical loops in scientific code, providing arguments to the advice that indicated numerical properties of the loop, such as the stride of a loop counter variable. Bodden and Havelund [5] created Racer, a race detection tool, using AOP and the *abc* system. In their work they also found that the existing set of pointcut designators was insufficient for their monitoring needs, and implemented their own new pointcut designators to specifically monitor locks and thus detect potential race conditions. Khaled et al. [12] used AspectJ for program monitoring, specifically for supporting program visualization. Hamlen and Jones [10] used AOP for the security monitoring of references, where security policies are checked in-line with the reference access. Bockisch et al. [4] describe VM support for dynamic join points, and this work may be able to provide the underlying capabilities needed for supporting time and data space dimension pointcut designators, and perhaps the probability/counting dimension as well.

A very nice formal framework for *Monitor-Oriented Programming* was detailed in [7]. This work describes the monitoring task in high-level formal notations, and demonstrates how AOP can be used to provide a rigorous framework for building runtime verification analyses. However, from the perspective of monitoring instrumentation, the MOP work in some sense *skipped* the hard part, by simply implementing their ideas on top of AspectJ. This means that all of the ideas are limited *in practice* to be usable only at the level of detail AspectJ provides: method call/return, field access, and a few other program events. Our contribution in this paper is relatively orthogonal to the MOP work; enabling MOP to use our extensions would produce a very powerful monitoring and verification framework.

7. CONCLUSION

This paper presented ideas for extending the normal AOP concepts to support the full range of runtime monitoring needs. We described two new pointcut designators that operate at a finer level detail over the base program's code, and we proposed that AOP support dimensions of weaving other than the source code: data, time, and sampling. We believe that these and other new ideas for aspect weaving will serve to enable AOP to be used for a large number of program monitoring tasks. This will move runtime monitoring from being dependent on highly technical instrumentation requirements to being generally available to developers who

need particular monitoring tasks. Although not addressed in this paper, we also imagine that these new dimensions of weaving, and added detail of code-dimension weaving, will be useful for other purposes that AOP supports. The ideas of different dimensions also open up new realms of thinking about dynamic weaving and runtime support, and other parts of "typical" AOP frameworks.

Many of our ideas are still very preliminary and need not only implementation, but also experimentation and refinement. We are continuing to work on implementing various pointcut designators, in particular more of the code-based designators. Another direction that needs investigation is improving the performance of monitoring advice, since controlling the overhead cost is generally a real concern in program monitoring. Ongoing work (e.g., [3]) that finds new ways to optimize advice execution may help make detailed runtime monitoring even more efficient and usable in the future. New mechanisms for making reflective information easier and faster to obtain in the advice code will also be needed, perhaps generating more static data during compilation so that advice can quickly access the data without costly runtime searches.

8. REFERENCES

[1] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. *SIGPLAN Not.*, 36(5):168–179, 2001.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Abc: an Extensible AspectJ Compiler. In *AOSD '05: Proc. 4th Int'l Conf. on Aspect-Oriented Software Development*, pages 87–98, New York, NY, USA, 2005. ACM.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *PLDI '05: Proc. 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 117–128, New York, NY, USA, 2005. ACM.

[4] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD '04: Proc. 3rd International Conference on Aspect-Oriented Software Development*, pages 83–92, New York, NY, USA, 2004. ACM.

[5] E. Bodden and K. Havelund. Racer: Effective Race Detection Using Aspectj. In *ISSTA '08: Proc. 2008 Int'l Symposium on Software Testing and Analysis*, pages 155–166, New York, NY, USA, 2008. ACM.

[6] J. Bonér and E. Kuleshov. Clustering the Java Virtual Machine using Aspect-Oriented Programming. In *Proc. AOSD 2007 Industry Track*, 2007.

[7] F. Chen and G. Roşu. MOP: an Efficient and Generic Runtime Verification Framework. In *OOPSLA '07: Proc. 22nd ACM SIGPLAN Conf. on Object Oriented Programming, Systems, and Applications*, pages 569–588, New York, NY, USA, 2007. ACM.

[8] R. Dyer and H. Rajan. Nu: a Dynamic Aspect-Oriented Intermediate Language Model and Virtual Machine for Flexible Runtime Adaptation. In *AOSD '08: Proc. 7th Int'l Conf. on Aspect-Oriented Software Development*, pages 191–202. ACM, 2008.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[10] K. W. Hamlen and M. Jones. Aspect-Oriented In-lined Reference Monitors. In *PLAS '08: Proc. 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 11–20, New York, NY, USA, 2008. ACM.

[11] B. Harbulot and J. R. Gurd. A Join Point for Loops in AspectJ. In *AOSD '06: Proc. 5th Int'l Conf. on Aspect-Oriented Software Development*, pages 63–74, New York, NY, USA, 2006. ACM.

[12] R. Khaled, J. Noble, and R. Biddle. InspectJ: Program Monitoring for Visualisation using AspectJ. In *ACSC '03: Proc. 26th Australasian Computer Science Conference*, pages 359–368, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proc. of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[14] K. Kiviluoma, J. Koskinen, and T. Mikkonen. Run-time Monitoring of Architecturally Significant Behaviors Using Behavioral Profiles and Aspects. In *ISSTA '06: Proc. Int'l Symposium on Software Testing and Analysis*, pages 181–190. ACM, 2006.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug Isolation Via Remote Program Sampling. *SIGPLAN Not.*, 38(5):141–154, 2003.

[16] E. Metz, R. Lencevicius, and T. F. Gonzalez. Performance Data Collection Using a Hybrid Approach. *SIGSOFT Softw. Eng. Notes*, 30(5):126–135, 2005.

[17] A. Nusayr and J. Cook. Extending AOP to Support Broad Runtime Monitoring Needs. In *Proc. 2009 Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE)*, 2009. to appear.

[18] M. Palankar and J. E. Cook. Merging Traces of Hardware-Assisted Data Breakpoints. In *WODA '05: Proc. 3rd International Workshop on Dynamic Analysis*, pages 1–7, New York, NY, USA, 2005. ACM.

[19] H. Rajan. A Case for Explicit Join Point Models for Aspect-Oriented Intermediate Languages. In *VMIL '07: Proc. 1st Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms*, page 4, New York, NY, USA, 2007. ACM.

[20] H. Rajan and K. Sullivan. Aspect Language Features for Concern Coverage Profiling. In *AOSD '05: Proc. 4th Int'l Conference on Aspect-Oriented Software Development*, pages 181–191, New York, NY, USA, 2005. ACM.

[21] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proc. of the 31st International Symposium on Computer Architecture (ISCA*, pages 224–235, 2004.