A Comparison of Compilation Techniques For Trace Monitors With Free Variables

Pavel Avgustinov, Julian Tibble, Oege de Moor
Programming Tools Group, University of Oxford

Abstract

A variety of different designs and optimisation strategies for trace monitoring have been proposed recently. Here, we examine tradeoffs in simplicity of implementation and expressiveness of supported patterns, briefly discuss the underlying data structures of two mainstream implementations, and provide a short evaluation of the effectiveness of memory optimisations.

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

General Terms Experimentation, Languages, Performance

Keywords Program monitoring, runtime verification, program analysis, aspect-oriented programming

1. Introduction

Trace monitors allow a programmer to write temporal patterns over the execution trace of a program. These patterns are automatically used to instrument the program so that, when the program is run, any match of the pattern triggers extra code to be executed.

Most current research can be seen to follow the paradigm of *monitor-oriented programming*, as proposed by Rosu *et al.* [3] In recent works — both our own system [1] and others [3–6] — a consensus has emerged that allowing free variables in the trace specifications is an indispensable feature for describing properties of cliques of interacting objects, and indeed it is in such contexts that trace monitoring is especially superior to other types of runtime verification (for instance, manual instrumentation).

Most popular implementations of trace monitors are based on finite state automata. In this work, we concentrate on some design choices and implementation strategies for such systems, with particular attention on the tradeoffs between ease of implementation, performance and completeness.

2. Types of leak-elimination

Trace monitoring systems work by translating the temporal pattern to an automaton at compile-time and then maintaining runtime data-structures that relate variable bindings to automaton states. Since doing that requires keeping references to runtime objects, it is all too easy to introduce space leaks into the program (that is, to keep alive objects that would otherwise have been garbage collected). A variety of strategies has been proposed in the past, ranging from the simple use of weak references [5] over a data structure that purges parts of itself when references expire [3] to a reasonably sophisticated analysis of variables per-state [1].

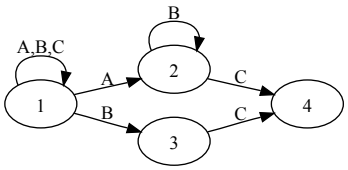

Figure 1. Monitor automaton for the pattern *(AB*|B)C*

In this analysis, we look for so-called *collectable variables*. Such variables *must* re-occur on *every* suffix that could complete a partially matched trace, and therefore it is sufficient to keep weak references to them — either they occur again and we can match against them, or expire and hence can't occur again. Moreover, any set of bindings referring to an expired collectable variable is *unsatisfiable*, as no suffix that would complete it is possible.

However, this strategy can break down for patterns using alternation. Consider for example the trace pattern $A(B|C)$, where the symbol A binds variables x and y, B binds x and C binds y. There is no single variable the expiration of which will guarantee unsatisfiability of the associated bindings after an A has been matched; thus, conventional leak elimination will fail. At the same time, it is easy to note that if *both* x and y expire, the match cannot complete.

We have implemented an analysis generalising this idea by annotating each automaton state with so-called *collect-sets*, which are just sets of variables. If *all* variables in such a set expire, we can drop all bindings that used to refer to these variables from the current state. This technique proved very effective on trace patterns with alternation, allowing us to increase performance up to ten-fold and reduce memory usage dramatically [2].

3. The effect of restricting variable binding

Restricting how monitor variables are allowed to be bound can simplify the implementation of a trace monitoring system. To see why, we will consider the automaton shown in Figure 1, for the regular expression *(AB*|B)C*. The symbol A binds x, B binds y, and C binds x and y.

A trace monitor for this pattern must store partial bindings (that is, bindings where not all monitor variables are bound to an object), because any potential match of the pattern begins with A or B, but neither of these symbols bind the full set of variables. We will see that this leads to *merging* and *splitting* bindings, and the need for *negative bindings*.

Table 1 shows the variable bindings associated with each automaton state after each of the events in the sequence $A(x = o_1)$, $B(y = o_2)$, $C(x = o_1, y = o_2)$.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-865-7/07/0010.

Event	1	2	3	4
$A(x = o_1)$	true	$x = o_1$	false	false
$B(y = o_2)$	true	$x = o_1$	$y = o_2$	false
$C(x = o_1,$	true	$x = o_1$	$y = o_2$	$x = o_1$
$y = o_2)$		$\wedge\, y \neq o_2$	$\wedge\, x \neq o_1$	$\wedge\, y = o_2$

Table 1. Variable bindings for each automaton state.

After the first two events, there are two independent variable bindings stored: $(x = o_1)$ on state 2, and $(y = o_2)$ on state 3. When the third event occurs these constraints are both *split*, because there is a non-deterministic choice. For example, for the first of these bindings, the choice is to take the C transition to state 4 and update the binding to $(x = o_1 \wedge y = o_2)$, or to reject the C transition and record the rejection by updating the binding to $(x = o_1 \wedge y \neq o_2)$. Inequalities in a variable binding are called *negative bindings*. Finally, the two constraints propogated to state 4 are identical, so they are *merged*.

A design decision made by the designers of JavaMOP [3] was to restrict the patterns accepted by their system: every word that matches a JavaMOP pattern must start with a symbol that binds *all* the monitor variables. With this restriction, variable bindings are completely independent — there is no splitting, no merging (at least if the automaton is determinised first, as otherwise it's possible to have bindings on different automaton states at the same time), and no need for negative bindings. This restriction simplifies the data structures used by the system, as we shall see in the next section.

4. Data structures for storing monitors

Trace monitors can have extremely large overheads when large numbers of variable bindings are stored simultaneously, unless these bindings are indexed. The tracematch system and JavaMOP both use multi-level trees (implemented using hash-maps) for this indexing, but they differ in the structure of the trees.

The tracematch system has a tree for each automaton state, and the leaves of each tree are variable bindings associated with that state.

In contrast, JavaMOP has a tree for each symbol, and the leaves of the tree hold (sets of) automaton instances, which can be thought of as simple counters. Each instance occurs in the appropriate leaf of all trees, to allow its quick retrieval no matter what symbol was matched. Since, as mentioned in the previous section, variable bindings are independent in the JavaMOP system, bindings are updated by looking up the relevant monitor instance and destructively updating the state variable.

This approach does force the restriction on variable bindings discussed in Section 3: The first symbol matched must bind *all* monitor variables, since it will create the instance and insert it into all trees; it is this restriction and the associated lack of constraint splitting and merging that allows the destructive updates mentioned above.

The draw-back is that there is no way to choose a leak elimination strategy per-state. Recall that with the alternative approach of annotating a single monitor instance with constraints, our space leak elimination determines how to handle each variable while it is on a given state, and this allows us to do a better job of detecting invalid constraints in many cases.

5. Performance and Future Work

One might expect that the approach of keeping one monitor instance for each of bindings might be more performant, particularly due to the absence of merging and splitting constraints, and that the generality of allowing partial matches carries an intrinsic cost.

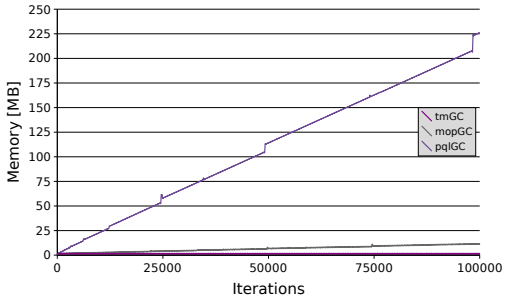

Figure 2. Memory usage for PQL, JavaMOP and tracematches

However, our comparative experiments with JavaMOP and tracematches do not confirm that. In fact, on many examples performance is close, while the more advanced per-state leak elimination strategy allows tracematches to keep a lower memory footprint.

A thorough discussion is beyond the scope of this document, but consider the memory graph shown in Figure 2, obtained by implementing the RV concern of *safe iteration* with different trace monitoring systems as a trace monitoring benchmark, as introduced by [1]. The top line is the memory usage of the PQL system [6], which pays no attention at all to space leaks, and correspondingly had terrible memory behaviour. The second line shows the footprint of JavaMOP [3], and clearly their use of weak references allows them to do significantly better; still, there is a clear upwards trend. Such a trend is completely absent in the final line, corresponding to tracematches with advanced per-state leak elimination — the memory usage stays practically constant throughout.

For reference, the runtimes of this benchmark are as follows: PQL takes 144s, as it is significantly hampered by many live objects. JavaMOP took 16.1s (using centralised indexing), and tracematches 12.1s. For a more thorough performance evaluation, the reader is referred to [2]; here, we just want to stress the importance of leak elimination and point out that the unrestricted generality of tracematch patterns doesn't come at a prohibitive cost compared to the JavaMOP implementation, which is optimised for the special case described in Section 3.

Still, it is clear that performance gains can be made by exploiting the lack of split and merge operations whenever we are dealing with fully-bound sets of variables. Indeed, we would like to develop a strategy for generating specialised code that makes use of this property at any state of the automaton for which it is guaranteed. Thus, we wouldn't sacrifice the generality of patterns by disallowing those that do not bind all variables upfront, but at the same time would get the performance benefits whenever possible.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA'05*, pages 345–364. ACM Press, 2005.

[2] Pavel Avgustinov, Julian Tibble, and Oege de Moor. Making Trace Monitoring Feasible. In *OOPSLA'07*. ACM Press, 2007.

[3] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *TACAS '05*. Springer, 2005.

[4] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. In *WODA '05*. ACM Press, 2005.

[5] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA'05*, pages 385–402. ACM Press, 2005.

[6] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *OOPSLA'05*, pages 365–383. ACM Press, 2005.