Feasibility of Combining and Optimizing Bytecode Traces.

Mathew Zaleski*

22nd April 2003

Abstract

The report summarizes the second experiment of my research project. It builds on the results of the first experiment in which a simulation of a trace-oriented JVM simulator called Jootch[1] was constructed and was used to run selected SPECjvm98 Java benchmarks. In this experiment bytecode traces selected by Jootch are combined together, hand-translated into C and compiled by `gcc`. The hand-coded C code does not model trace exits faithfully. Nevertheless, as a rough feasibility study, the results show that our proposal to combine traces into a dynamically identified unit of compilation may perform well.

1 Combining and Optimizing Traces

I would like to establish the feasibility of my suggestion that if traces are combined into a *combined compilation unit* (CCU) and optimized system performance will benefit. This report makes the first step, by demonstrating that a general-purpose optimizer can significantly improve the performance of traces selected from one small program.

We have found that a reasonable optimizer, `gcc`, has managed to exploit the multiple loop nesting structure of the program. Loop invariant code that started out in a single trace that formed the innermost loop of the program was hoisted outside the containing loop. The main result is that the optimizer speeds up the execution of the traces by a factor of two. This indicates that optimizing the CCU is likely to be worthwhile. The optimized, hand-coded C program derived from the traces runs about twice faster than a state-of-the-art JIT runs the original Java program.

*Research Proposal for DCS PhD $Revision: 2.1 $

[1] Jootch stands for Java Object-Oriented Trace Cache heuristic.

The rest of this report will be structured as follows. First the experiment will be described, then the sample program will be discussed. The trace generation carried out by Jootch will be described in great detail. Then the hand-translated C code will be presented and finally the assembler produced by gcc -O2 will be discussed.

1.1 Background and related work

The scientific background for this report is probably best summarized by the paper written in support of my PhD depth oral examination [3]. A much condensed background section appears in the first few pages of my research proposal [1]. This report also assumes some familiarity with the Jootch JVM simulator, which I have described in [2].

1.2 The Experiment

First, a small Java program is run by Jootch. The sample program, created for this purpose, has a multiply nested structure that is obscured by polymorphism. The bytecode traces identified by Jootch will be combined, by hand, into a single flow graph. This will be translated, by hand, into a C program. The hand-translation does not model the Java local variable array and operand stack or trace exits in detail – our aim at this early stage is to determine if the flowgraph of the linked traces can be dealt with by a general-purpose optimizer. The C program will be compiled and optimized by gcc. The resulting code will be run as well as examined.

1.3 Doubly Nested Java Program

Our small example program searches an array of objects for a particular query object. The source for our program appears in Figures 1 and 2. There are two interesting methods. `Trace2.loop` searches an array of `Objects` (called `otab`) for a query string (`q`). The equality check is made by sending the `equals`[2] message to each object in `otab` passing `q` as a parameter. Since all the objects in `otab` are instances of `String` this always results in a call to the `String.equals` method.[3]We have concocted, in a realistic way, an effectively monomorphic callsite. We have borrowed `String.equals` straight from Sun's JDK sources. It starts out with a few optimizations that check if the parameter is non-null, an instance of `String`, and if the comperand is the same length. If all these conditions are met `equals` loops to check whether each char of the comperand is the same as the receiver.

Our simple program can be thought of as a doubly nested loop that is obscured by virtual method invocation. In this view the program has an outer loop

[2]The `equals` method is defined by the java `Object` class.

[3]Figure lists the Java Foundation Class source for `String.equals`. (This is an extract from Sun's licensed JDK sources hence clean room Java implementors may not be permitted to view this code).

```
public class Trace2 {
    static String stab[] = {
        "xx",
        "yy",
        "yy",   //trace generate Trace1
        "123x", //trace generate Trace2
        "123y",
        "123z", //trace generate Trace3
        "a234", //trace generate Trace4
        "b234", //runs from cache
        "1234" //match
    };
    public int loop( Object[] otab, Object q ){
        for(int i=0; i< otab.length; i++){
            if ( otab[i].equals(q) ){
                return i;
            }
        }
        return -1;
    }
    public static void main( String[] args ){
        new Trace2().loop(stab, "1234");
    }
}
```

Figure 1: Trace2.java

```
/**
 * *** This code from Sun JDK sources ***
 * Compares this string to the specified object.
 * The result is <code>true</code> if and only if the argument is not
 * <code>null</code> and is a <code>String</code> object that represents
 * the same sequence of characters as this object.
 *
 * @param   anObject   the object to compare this <code>String</code>
 *                     against.
 * @return  <code>true</code> if the <code>String </code>are equal;
 *          <code>false</code> otherwise.
 * @see     java.lang.String#compareTo(java.lang.String)
 * @see     java.lang.String#equalsIgnoreCase(java.lang.String)
 */
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = count;
        if (n == anotherString.count) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = offset;
            int j = anotherString.offset;
            while (n-- != 0) {
                if (v1[i++] != v2[j++]) {
                    return false;
                }
            }
            return true;
        }
    }
    return false;
}
```

Figure 2: Source for `String.equals` from Sun's JDK. (Clean room developers and designers may not be permitted to study this code.)

in `Trace2.loop` and an inner loop in `String.equals`. Since there is a dynamic method invocation in between it is not trivial for a static compiler to exploit this structure. Once it is known that the `invokevirtual` in `Trace2.loop` is effectively monomorphic it is obvious to a human that the code is a doubly nested loop. If we wait until after traces have been generated perhaps this will obvious to gcc too.

1.4 Details of Trace Generation

The way in which traces will be selected depends on the value of HOT and the particular strings with which we initialize the `stab` array. For the purpose of discussion suppose we set the HOT threshold very low, say two. Let `q` be `"1234"`, as in Figure 2. We will set `otab to` contain a few strings that are different lengths than `q` and then a few more that are the same length as `q` and differ only in the last few characters. The point of this is to contrive `otab` in such a way so that the outer loop will become hot and be trace generated first, then the inner loop, then the two will be linked together by a two more traces. Figure 3 illustrates how basic blocks from the methods `Trace2.loop` and `String.equals` are arranged into three traces. Basic blocks starting with "`l`" (red in the figure) originate from the `loop` method and basic blocks starting with "`s`" (in blue) originate from `String.equals`.[4] (Figures 4 and 5 list the bytecode and control flow graphs for each method.)

Next we will describe in detail how each trace is generated and then we can discuss the resulting traces and then how the trace might be combined. Table 1 lists the role each element of `otab` plays in our contrived example. Basic blocks as illustrated by Figure 3 are often referred to.

1.4.1 Blow by blow trace generation

Is this level of detail interesting to anyone? If no one notices this sentence the answer must be no.

Trace1 The first two strings ("`xx`" and `"yy"`) make the outer loop hot and the third (`"zz"`) is trace generated. Since the strings differ in length from the query the inner loop is never entered. The characteristics of `"zz"` thus determine the code generated in Trace1. The call to `equals` is guarded and then the destination method, `String.equals` is inlined. Since `"zz"` differs in length from `q` the method returns through block `s8`. (This return does not need to be guarded as it matches a call in the same trace.) Trace1 ends when trace generation notices that a cycle would be formed by the reverse branch at the bottom of block `l4`.

Trace2 Now that Trace1 has been trace generated it is dispatched.

[4] Yes, this is inconsistent. These bb's should have been named starting with "s".

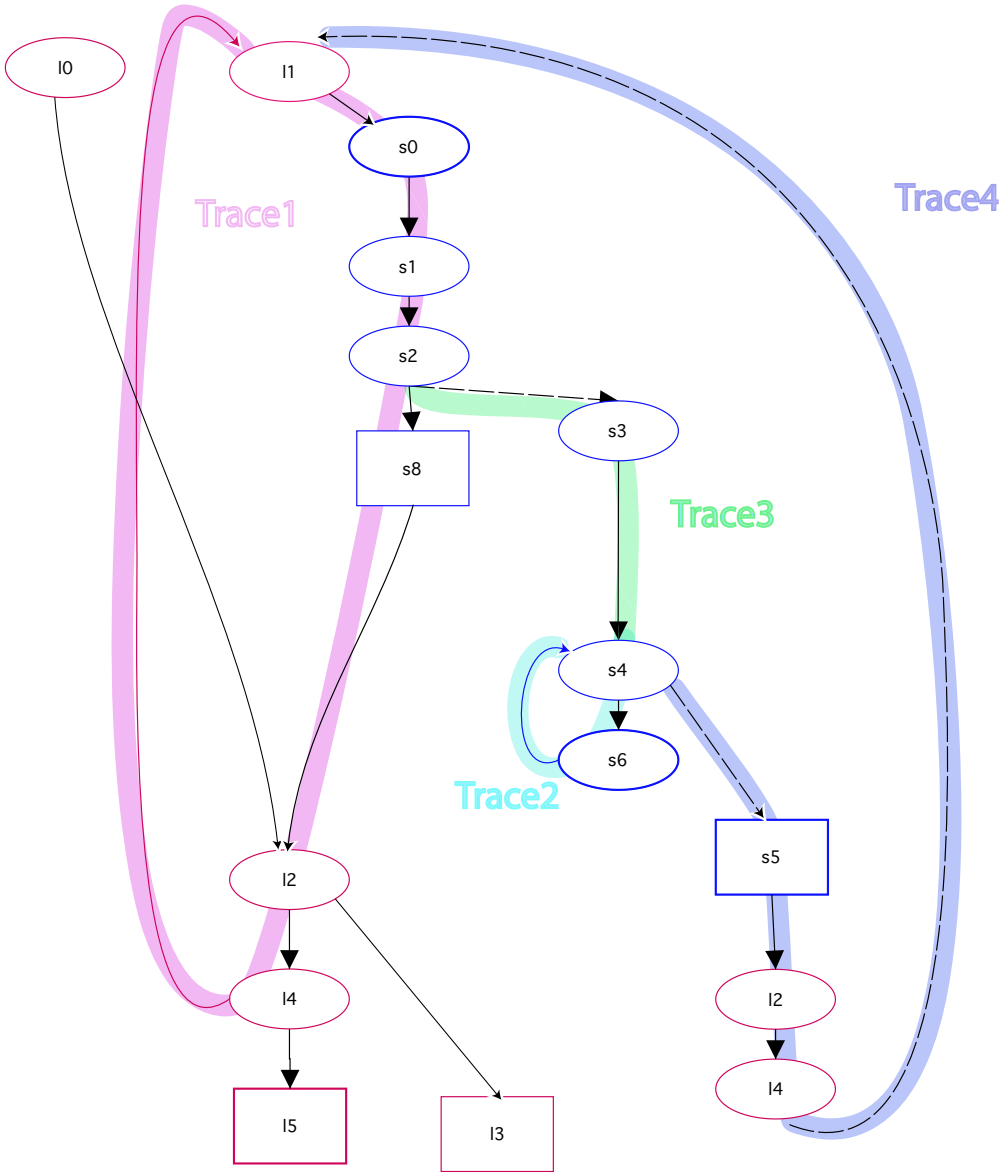

Figure 3: Illustration of traces in cache for Trace2. Basic block labels correspond to the labels in Figures 4 and 5.

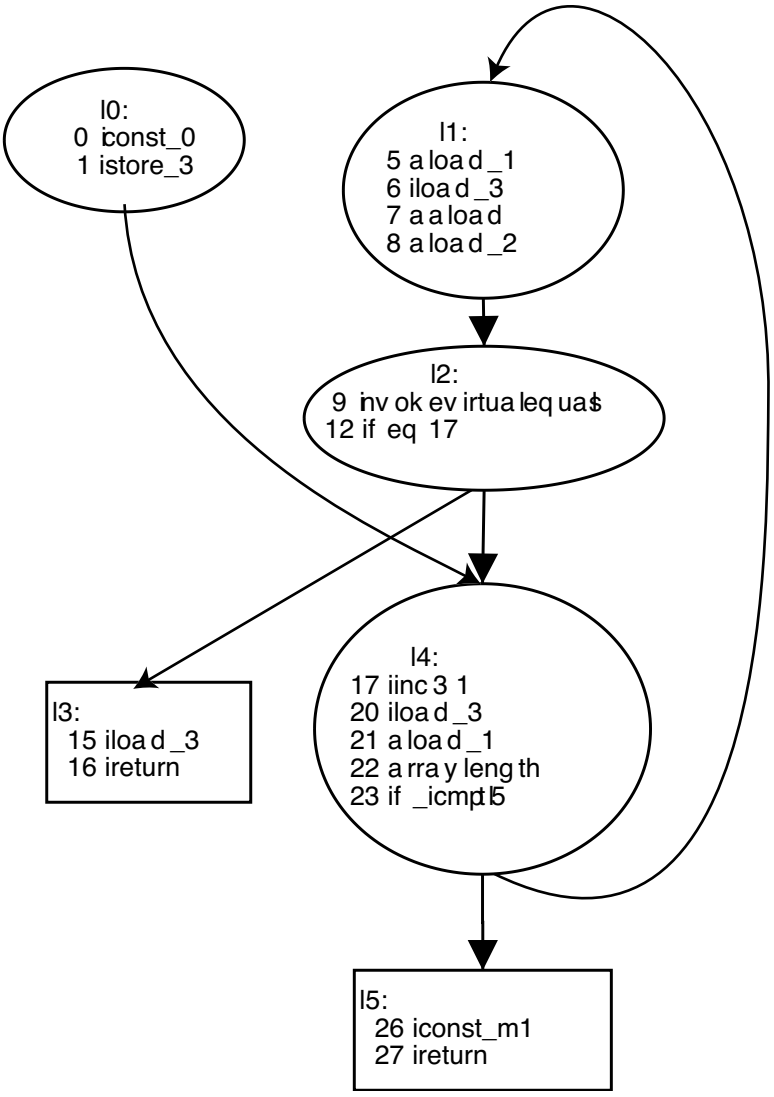

Figure 4: Original `Trace2.loop` control flow graph annotated with bytecode contained in each basic block.

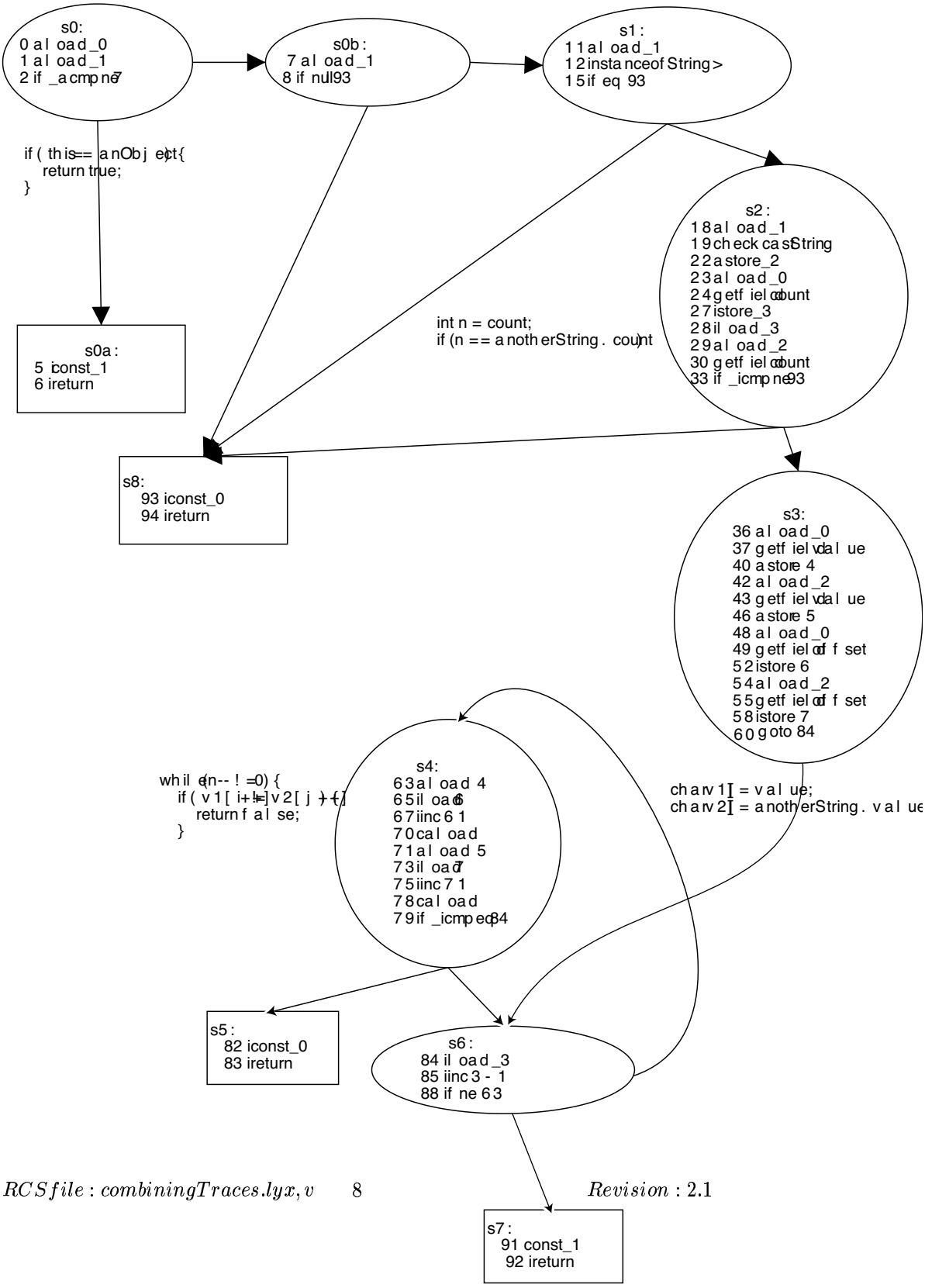

Figure 5: `String.equals` control flow graph and bytecode.

otab element	string value	comment
0	xx	Warms up outer loop l4 latch
1	yy	Fully warms up l4 latch
2	zz	Trace generates Trace1
3	123x	Dispatches Trace1. Warms up trace exit from Trace1 (from s2) and trace generates Trace2 (inner loop)
4	123y	Fully warms trace exit from Trace1 (from s2) and warms trace exit from Trace2 (s6).
5	123z	Trace generates Trace3 connecting Trace1 and Trace2. Fully warms trace exit from Trace2 (at s6).
6	a234	Trace generates Trace4 connecting Trace2 and Trace1.
7	b234	Runs from trace cache.

Table 1: Details of otab data and trace generation of Trace2.

The string s[3] ("123x") is the same length as q and so warms the trace exit from Trace1 at the conditional trace exit corresponding to branch that at the end of block s2. Interpretation thus resumes at block s3 of String.equals.

As this String is the same length as q the inner loop (blocks s4 and s6) of String.equals iterates twice (for "12") becomes hot, and then is trace generated on its third iteration. Trace2 contains just this loop.[5] Trace2 is dispatched with the remaining characters of the string (just "x" in this case). Since x differs from the last char of the query the conditional trace exit at the bottom of s4 is taken.

Trace3 The next string in otab, s[4] "123y" causes the trace exit from Trace1 to become hot. Hence Trace3 is trace generated for "123z". This is a short trace starting from the now hot trace exit from Trace1 to the target of the rearward branch that begins Trace2. All three traces link together.

Interpretation resumes but immediately reaches the head of Trace2 and so it is dispatched. Trace2 executes as far as the conditional trace exit at the bottom of s4 which becomes hot as a result. Interpretation resumes again in s5, returns and reaches the head of Trace1.

Trace4 Trace1 links to Trace3 which links to Trace2 so s[6] "a234" executes in the trace cache to the hot conditional trace exit from Trace2. Trace generation is entered for the last time in this example. Trace4 extends from Trace2 to the latch of the outer loop in Trace1 (originally block l4). Trace4 includes the return guard matching the call guard in Trace1.

[5] Note that this is an opportunity for return guard problems in a bigger program.

All remaining[6] `String` entries in `otab` execute from the trace cache until a match occurs and the program exits.

1.5 Issues with the Generated Traces

On the surface the trace selection heuristic has performed its job admirably. The call to `equals` has been guarded and inlined. However, there are a couple of characteristics of the traces illustrated in Figure 3 that may prove challenging.

1. Return Guard Problem. In a larger program we should expect `String.equals` to be called from many places. Hence one would expect that Trace2 might be linked to by many traces. Then the return guard in Trace3 would often fail.

2. Code replication. Blocks `l2` and `l4` appear in both Trace1 and Trace3. Obviously a more complicated if-then-else (one with more conditional trace exits) statement could cause more replication yet.

3. Complex flowgraph. The double rearward branch in Figure 3 does not look very optimizer-friendly. Particularly worrisome is the possibility an optimizer would fail to hoist invariant code from the innermost (`String.equals`) loop.

So far, we have no solution for the return guard problem. Hopefully we will have an opportunity to investigate it later in this project. Code replication is not clearly a problem – it might also be a benefit when it helps achieve better branch prediction and/or better utilize instruction cache prefetch bandwidth.

1.6 Hand coded traces in C

A trace-based JIT has the opportunity to create a combined control flow graph as traces link together in the trace cache. We anticipate that a heuristic can be found that will detect when interesting new loop structures emerge. Our proposed scheme would then translate the traces into some intermediate representation producing a combined compilation unit. To form an initial impression of the code quality of the code that could be produced by optimizing this new compilation unit we will hand-code, in C, a function that does more-or-less the same thing. Then we will compile and optimize the C using gcc and examine the resulting code. This program appears in Figure 6

We are curious[7] to learn if a realistic optimizer cleans up the loop invariant code in the innermost loop. If it does there is some reason to expect that our proposed technique may work. In any case the performance of our hand coded C can serve as a target to strive towards.

Figure 7 shows the resulting assembler. The flowgraph created by Jootch is clearly evident. Figure 8 focuses on the innermost loop before and after optimization.

[6] Note that nulls or instance of object other than String will cause trace exits from Trace1.
[7] anxious even

```
  typedef struct string {     | doloop(string otab[], int lotab, string q){
    char *s_ptr;              |   int i_otab;
    int s_len;                |   string s;
  } string;                   |   int i_str;
                              |   char *s1;
string otab[] = {             |   char *q1;
  {"xx", 2},                  |   i_otab = 0;
  {"yy", 2},                  | trace1: //******** trace1 ************  <--+
  {"1234", 4}                 |    s.s_ptr = otab[i_otab].s_ptr;         //|
};                            |    s.s_len = otab[i_otab].s_len;         //|
int otab_len = 3;             |    if ( s.s_len == q.s_len ){            //|
int *stack = 0;               |      *stack++ = 0; //push false            |
                              |      goto trace3;  //linked trace exit     |
main(){                       |    }                                     //|
  stack = malloc(42);         | if ( *(--stack) == 1) return -1;         //|
  string q;                   |   i_otab++;                              //|
  q.s_ptr = "1234";           |   if (i_otab<=otab_len) goto trace1; // ---+
  q.s_len = 4;                |   goto trace5;     //linked trace exit
  doloop(otab,3,q);           | trace2: /********* trace2 ************/ <--+
  return *(stack-1);          |   if (s.s_ptr[i_str] != q.s_ptr[i_str]){ //|
}                             |     goto trace4;  //linked trace exit      |
                              |    }                                     //|
                              |   i_str++;                               //|
                              |   if (i_str<=q.s_len) goto trace2;  // ----+
                              |   goto trace5;   //linked trace exit
                              | trace3:/********* trace3 ************/
                              |   s1 = s.s_ptr;
                              |   q1 = q.s_ptr;
                              |   i_str=0;
                              |   goto trace2;   //linked trace exit
                              | trace4: /********* trace4 ************/
                              |   *stack++ = 0; //push false
                              |   if ( *(--stack) == 1) goto trace5;
                              |   i_otab++;
                              |   if (i_otab<=otab_len) goto trace1;
                              | trace5: /********* trace5 ************/
                              |   *stack++ = 1; //push true
                              |   //return guard here
                              |   if ( *(--stack) != 1){
                              |     return -1; //trace exit
                              |   }
                              |   *stack++ = i_otab; //push true
                              |   return; //exit from trace cache
                              | }//
```

Figure 6: A Hand coded highly simplified combined trace cache for Trace2

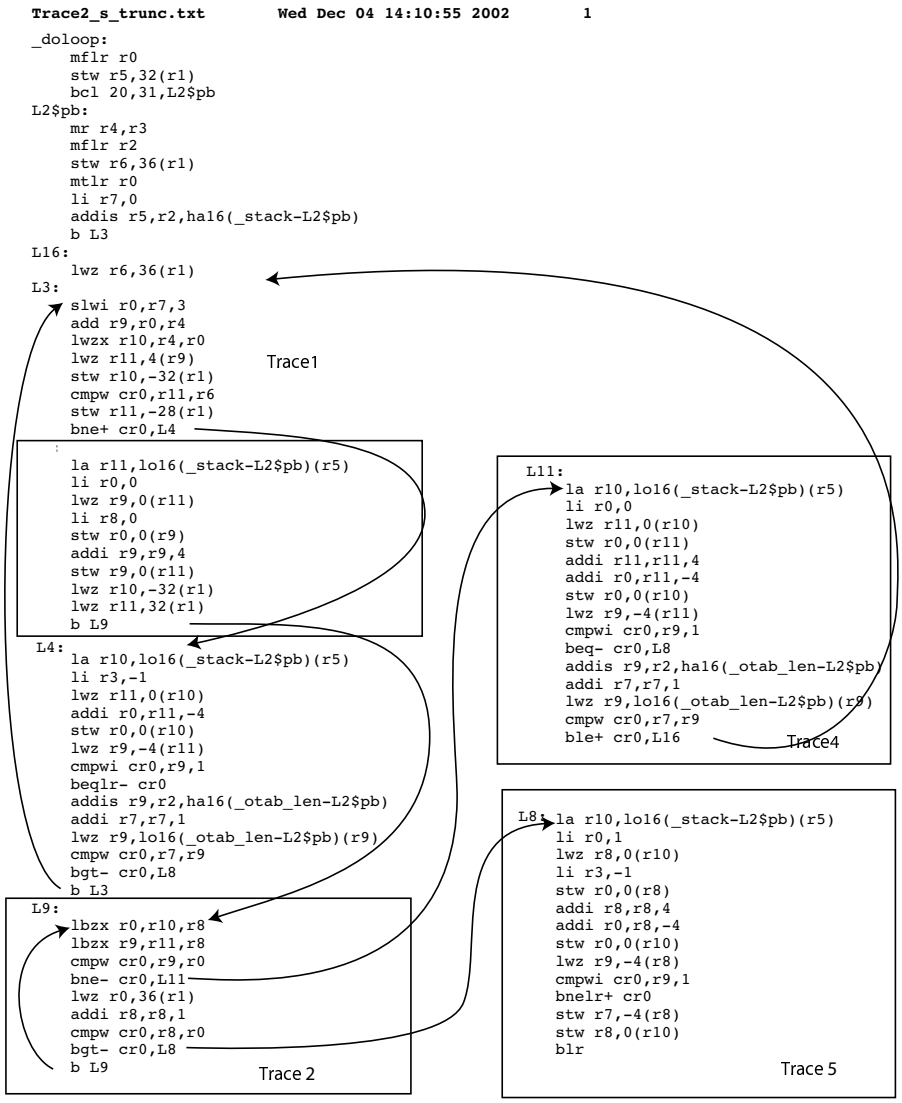

Figure 7: Output of gcc -O2 -S on hand coded C

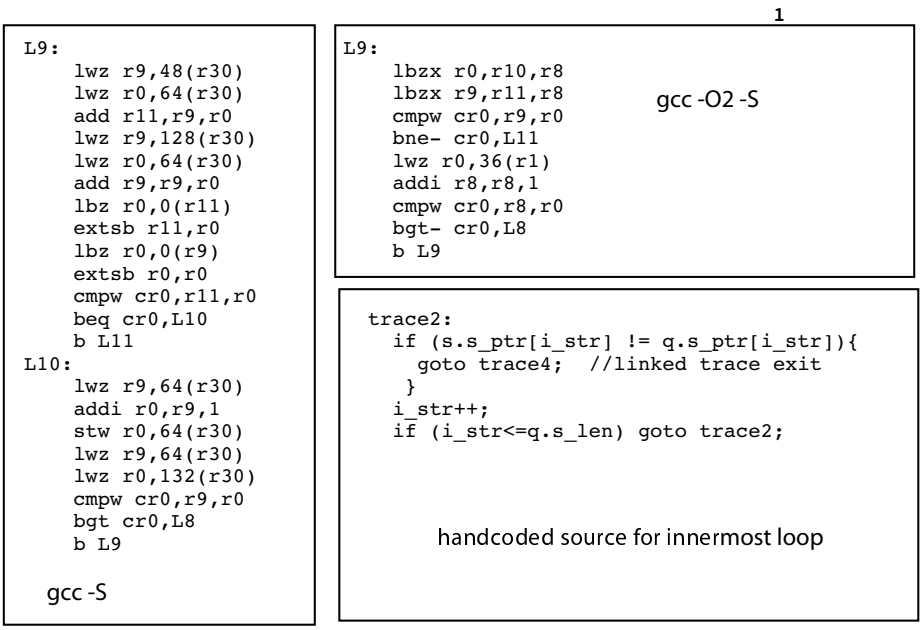

Figure 8: Comparison of inner loops

1.6.1 Simplifications in Hand Coded Version of Traces

Our hand-coded C does not attempt to model the runtime environment of the trace cache with much fidelity. This is not conservative since ignoring traceexits in the way we have done may free the optimizer to remove operations that a better model would not. The C code in Figure 6 does not model local variable array, operand stack or trace exits This includes call and return guards. Heap data is modeled as static or malloc'd data.

1.7 Performance Results

The demonstration shows that our idea of combining traces has some potential to perform well. The hand coded C runs slightly more than twice as fast as Sun's Hotspot JIT can execute the original Java program.

1.7.1 Performance of hand coded C

When we call `doloop` 10 million times on a 500MHz PowerPc[8] the `-O2` optimized version of our hand coded C runs for 2.0 (elapsed) seconds whereas the unoptimized code runs for 5.2 seconds time. Compiling the code on the same

[8] Apple PowerBook G4 running Mac OSX 10.2

hardware with gcc version 3.1 at O2 required about one third of a second elapsed compilation time.

1.7.2 Performance of Production jit and Trace2

To execute `Trace.loop` ten million times (with the equivalent `otab` array to the one in `Trace2.c`) Sun's Hotspot JVM required about 4.3 elapsed seconds on the same hardware. (This is about ten times faster than with the JIT turned off.)

We would like to evaluate how good code the Sun Hotspot JIT generated into its code cache. Unfortunately there is no documented user interface to cause Hotspot to dump its generated code. The only clue we obtained (via `java -XX:+PrintCompilation`) was a log of compiled and inlined methods. From this we surmise that Java HotSpot Client VM (build 1.3.1_03-69, mixed mode) did not inline `String.equals` into `Trace2.loop`.

1.8 Discussion of Results

The overall impression of this experiment is that our scheme for combining traces should be pursued further. The results suggest that our technique may achieve make it possible for classical optimization techniques to make inter-procedural improvements despite the presence of a polymorphic callsite obscuring the multiply nested structure of our sample program.

Figure 8 shows that the inner loop, originally trace-generated by Jootch as bytecode Trace2, has been cleared of loop invariant code. The inner loop is less than half as many instructions long in the O2 optimized version.

It should be reiterated that the simplifications made concocting our hand coded traces ignore the support of trace exits. Though we hope we will be able to implement trace exits efficiently this has not been worked out and might be extremely difficult. This means that our hand-coded translation of traces may be better than a realistic JIT can achieve. Nevertheless, is worth noting that Trace2, the innermost loop of the combination compilation unit, contains only one trace exit. This suggests that the a better model of trace exits may not make that much difference in this particular case.

References

[1] Mathew Zaleski. A better way of running object-oriented bytecode? a research proposal. http://www.cs.toronto.edu/~matz/prop.pdf, April 2003.

[2] Mathew Zaleski. Jootch, a simulation of bytecode trace selection and creation in a JVM. http://www.cs.toronto.edu/~matz/traceCreation.pdf, April 2003.

[3] Mathew Zaleski. Trace-based dynamic compilation for object-oriented programming systems. http://www.cs.toronto.edu/~matz/background.pdf, April 2003.