Phase Detection using Trace Compilation

Christian Wimmer Marcelo S. Cintra Michael Bebenita Mason Chang
Andreas Gal Michael Franz
Department of Computer Science
University of California, Irvine
{cwimmer, mcintra, mbebenit, changm, gal, franz}@uci.edu

ABSTRACT

Dynamic compilers can optimize application code specifically for observed code behavior. Such behavior does not have to be stable across the entire program execution to be beneficial for optimizations, it must only be stable for a certain program *phase*. To specialize code for a program phase, it is necessary to detect when the execution behavior of the program changes (*phase change*). Trace-based compilation is an efficient method to detect such phase changes. A trace tree is a collection of frequently executed code paths through a code region, which is assembled dynamically at run time as the program executes. Program execution tends to remain within such a trace tree during a stable phase, whereas phase changes cause a sudden increase in side exits from the trace tree. Because trace trees are recorded at run time by observing the interpreter, the actual values of variables and expressions are also available. This allows a definition of phases based not only on recurring control flow, but also on recurring data values. The compiler can use constant values for variables that change their value rarely and rely on phase detection to handle the case when the variable value actually changes. Our evaluation shows that phase detection based on trace trees results in phases that match the intuitive expectation of a programmer and that are also useful for compiler optimizations.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Languages, Performance

Keywords

Java, phase detection, data flow, trace compilation, just-in-time compilation, optimization

PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.
http://doi.acm.org/10.1145/1596655.1596683

1. INTRODUCTION

The success of the programming language Java has propelled virtual-machine based program execution into the mainstream. Java's execution model allows for continuous just-in-time (JIT) compilation and profile-guided optimization. This enables aggressive dynamic optimizations [1] that specialize the machine code towards the current execution *phase* of the application. Problems arise when the program phase changes after optimizations have been applied: an optimization that has improved the performance of the first execution phase might decrease the performance of subsequent phases. To avoid this, the virtual machine must detect such phase changes, revert the optimizations, and then apply a different set of optimizations.

Phase detection splits the application lifetime into several non-overlapping intervals where execution behavior is similar within a given interval, but different from adjacent intervals. Such an interval is called a *stable phase*. Between two stable phases, there can be transitioning code that is executed rarely and therefore not part of any stable phase.

Figure 1 shows an example of a local phase change inside a method. It contains the control-flow graph of a loop that has an if-else-condition in the loop body. Assume that only the if-branch is frequently executed for a while. The JIT compiler can optimize for this code path. When the execution phase changes and the execution frequency of the else-branch increases, the phase is no longer stable and the optimization no longer beneficial. Assume that after a short period of instability only the else-branch is frequently executed. The JIT compiler can now optimize the else-branch and treat the if-branch as code not affecting performance.

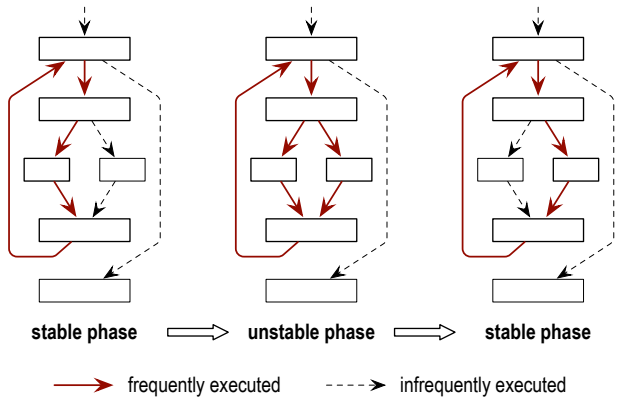


Figure 1: Example for a phase change inside a loop.

Traditionally, JIT compilers operate on a per-method basis, i.e., they compile one method at a time. This makes it difficult for the compiler to distinguish important parts of the method from rarely executed code, e.g., code for exception handling. To overcome this problem, we use *trace-based* compilation [8] where the compilation follows only execution paths that are frequently taken at run time. Machine code is generated only for these paths. If a rare code path is executed that is not covered by a compiled trace, a *side exit* is taken from the compiled trace and execution continues in the interpreter. In other words, traces are an implicit speculation on control flow.

We believe that traces are also ideal for phase detection. The execution frequency of traces and the number of side exits are good estimates for the application behavior. Since compiled traces cover frequently taken execution paths, execution that follows these compiled traces implies that the program is in a stable phase. A sudden increase of side exits indicates a phase change. Phase detection based on this data is simpler compared to previous approaches based, e.g., on profiles of all conditional branches [14], program counters [4], or cache miss rates [9].

Traces do not only cover the control flow, but also the data flow of the one prototypical trace execution that was recorded. The JIT compiler can freely decide what to do with this information; it can either ignore it, which means that no optimizations are performed based on actual data values, or assume that some values are the same in subsequent runs. We use the information to define phases not only by control-flow paths that are frequently taken, but also by data values that are stable for a certain timeframe.

To the best of our knowledge, we are the first to perform phase detection based on specific data values. In summary, this paper contributes the following:

- We perform local phase detection based on trace trees.
- We define phases not only on control flow, but also on specific associations between data values and variables.
- We use phase information for guiding optimizations of the trace compiler.
- We present a prototype implementation and evaluation in a research Java VM.

2. TRACE COMPILATION

Trace compilation uses a different paradigm than most other just-in-time (JIT) compilers. Instead of adhering to principles derived from traditional static compilers, trace compilation is only applicable for dynamic compilers because it is based on run-time profiling information. Execution starts in the interpreter. It keeps track of frequently executed "hot" parts of the program, which are only loops in our current implementation. We detect hot loops by incrementing backward branch counters. When a counter exceeds a certain threshold, the loop is considered worth for compilation.

In the next loop iteration, the interpreter records the execution of each bytecode using the *trace recorder*. The trace recorder constructs the trace compiler's intermediate representation. Aside from containing the opcode, operands, and the current program counter, each instruction also contains its current value, as observed during execution. This is the main difference to traditional *abstract interpretation* where no actual values are available. Recording stops when the interpreter reaches the loop header again. A trace can be seen as one prototypical execution of the loop, which is then compiled into machine code that is suitable for the succeeding loop iterations. When the interpreter reaches a position where a compiled trace starts, it invokes the machine code of the trace. Figure 2 illustrates trace recording and compilation.

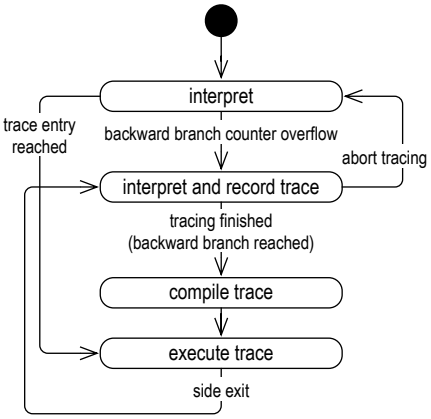


Figure 2: Trace recording and compilation.

A trace is a linear instruction sequence without branches. Conditional branches inside the original loop are recorded as *guard* instructions. They ensure that the control flow during machine code execution still follows the original recorded trace. A guard checks the condition of the original branch, and takes a *side exit* to the interpreter when the condition is different. The interpreter starts trace recording again at the side exit point and records instructions until the loop header is reached again. This results in a *trace tree* that covers all frequently taken paths through a loop. If the loop header is not reached during trace recording, it is aborted after a certain time.

Trace recording does not stop at method calls, but follows the execution of the called method. This is similar to partial method inlining of traditional compilers. For virtual method calls, a guard instruction is inserted to check that the called method is the method seen during recording. Effectively, trace compilation thereby performs dynamic method specialization. No control-flow graph is built during compilation. Because of the linear structure of traces, compilation is also simpler compared to traditional compilers. Control flow does not merge inside a trace, but only at the loop header, which allows optimizations to be performed mostly without complex data-flow analyses.

2.1 Example

Figure 3 shows the main computation loop of the Series benchmark, which is a part of Java Grande benchmark suite [3]. It performs numerical integration of the function defined in the method `thefunction()` by dividing the interval between `x0` and `x1` into `steps` trapezoids and summing up the area in the variable `result`. To improve readability, we removed one parameter of the methods and do not show the code before and after the loop. However, this does not affect the execution behavior of the benchmark.

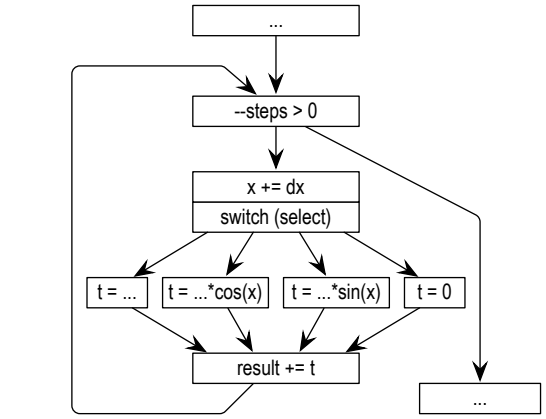
```
double integrate(double x0, double x1,
                 int steps, int select) {
  double x = x0;
  double dx = (x1 - x0) / steps;
  double result = ...
  ...
  while (--steps > 0) {
    x += dx;
    result += thefunction(x, select);
  }
  ...
  return result;
}

double thefunction(double x, int select) {
  switch (select) {
    case 0: return pow(x+1, x);
    case 1: return pow(x+1, x) * cos(x);
    case 2: return pow(x+1, x) * sin(x);
  }
  return 0;
}
```

Figure 3: Java Grande Series example.

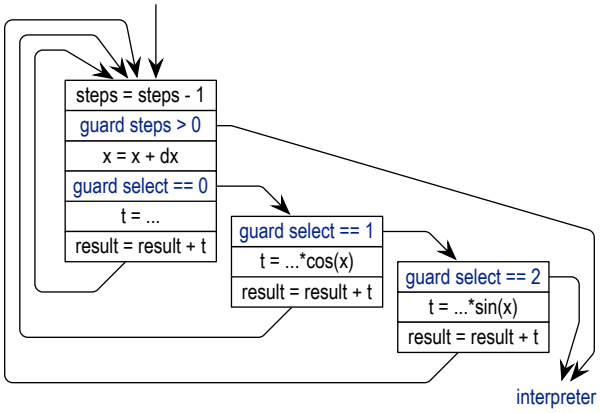
When compiling the method `integrate()`, a traditional compiler would first build a control-flow graph as shown in Figure 4(a). Assume that the compiler decided to inline the method `thefunction()`, so that the control flow for the `switch` statement is contained in the loop. Machine code is also generated for the complex code parts before and after the loop, although this code is executed much less frequently than the loop body.

In the trace-based system, compilation does not start at the method entry, but at the loop header. The first loop iterations are interpreted, and the interpreter increments a backward branch counter. When the counter exceeds a certain threshold, records the trace. A trace is a list of intermediate representation instructions for the compiler. The conditional branch at the loop header checks if the variable `steps` is less than or equal to 0 and exits the loop in this case. During trace recording, the branch is not taken. The recorded trace is only correct as long as `steps` remains greater than 0, so the trace recorder appends a guard instruction that checks for this condition. When the guard fails, a side exit from the trace is taken and execution continues in the interpreter. In other words, guards replace control-flow instructions that are used by traditional compilers.

The `switch` statement inside the loop is also compiled into guard instructions. During the first loop iterations, the value of `select` is 0, so the first trace contains only this code path. A guard ensures that a side exit is taken for other values of `select`. After some time, the method is called with the `select` value 1. The guard fails, and execution continues in the interpreter. Because the path starting at this side exit reaches the loop header again, the interpreter also records a trace for it. This second trace is attached to the side exit of the first trace. The two traces together form a trace tree and are compiled together. The same happens for the `select` value 2, which leads to a third trace to be recorded. Figure 4(b) shows the resulting trace tree. Note that the default path of the `switch` statement is never executed and therefore neither recorded nor compiled.



(a) Control-flow graph



(b) Compiled trace tree

Figure 4: Control-flow graph and traces for example.

3. PHASES BASED ON CONTROL FLOW

Phase detection is the abstract problem of finding parts of an execution profile where the behavior of the application is similar. Hind et al. introduced two parameters that must be specified [10]: *granularity*, which defines how the profile is split into chunks for comparison, and *similarity*, which defines how the chunks are compared. A phase is then a sequence of chunks where the profile is similar. We define the two parameters as follows:

- Granularity: We operate on the granularity of traces. Because we do not want to interrupt the optimized machine code of a trace, we check for phase change only when a side exit is taken and execution continues in the slower interpreter anyway. Checking at every side exit would lead to a too short measurement interval where no stable phases could be detected, so several side exits are combined to a *chunk*. The *chunk size* is a configurable but fixed number of side exits that must be taken before the check is performed.

- Similarity: We define similarity based on the execution counters for traces (called *trace counters*) and side exits (called *side exit counters*). Each backward branch in a trace and each side exit has a distinct counter. A similarity value is computed from these counter values.

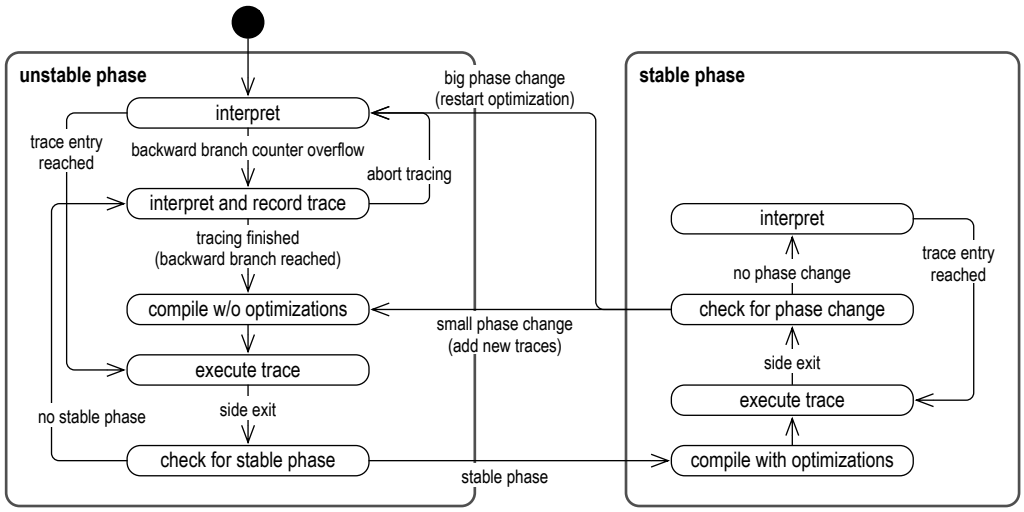


Figure 5: Phase detection integrated into trace recording.

The management of the counters and the computation of the similarity values are performed independently for each trace tree. This means that each trace tree has its own phase state. This is called *local phase detection* [4]. A global phase state [10, 14] would not be useful in our system because we use the phase information for guiding compiler optimizations, which operate only at the level of trace trees. The information that the application behavior has changed from executing one trace tree to another trace tree cannot be used for compiler optimizations within one trace tree. However, the trace counters and side exit counters could also be used for global phase detection by applying the similarity formula on the counters of all trace trees.

3.1 Structure of the Phase Detection System

Figure 5 shows how phase detection is integrated into the recording and compilation of traces. The first states are equivalent to the basic execution system presented in Figure 2. Execution starts in the interpreter, and when a backward branch counter crosses a certain threshold, a trace is recorded and compiled. For this compilation, complex compiler optimizations are disabled so that compilation is fast. If there is more than one frequently executed path through the loop, these traces are also recorded and attached to the corresponding side exits. This results in the trace tree being built fairly quickly. After a short time, no more new side exits are taken, so no trace recording occurs anymore.

Trace counters and side exit counters are incremented while the trace is executed and used to compute the similarity value. When this value crosses a certain threshold, a stable phase is detected. Because it is now unlikely that new traces are added to the trace tree, we compile the trace tree with compiler optimizations enabled and execute the optimized machine code. The counter values are still checked regularly, and a phase change is detected when the similarity value drops below the threshold.

Depending on the kind of phase change, either new traces are added to the trace or the whole trace tree is discarded. In most cases, a phase change means that new traces are added to the existing trace tree (small phase change). Only if the trace tree contains traces that were never executed in this phase, a special handling is performed: the whole trace tree is deleted and all trace recording starts anew (big phase change). An unexecuted trace means that the corresponding code path is no longer needed. By deleting the trace tree, we ensure that this code path is eliminated. This reduces not only the size of the machine code, but also improves the code quality of the remaining traces because it enables optimizations. For example, variable definitions that were only used in the deleted traces can be removed from the remaining code (dead code elimination), or code that is now loop invariant can be moved into the loop header (loop invariant code motion). Going back completely to the interpreter is also important for phase detection based on data flow, as shown in Section 4.

3.2 Computation of Similarity

The computation of the similarity value requires counter values of the current chunk and the previous chunk. Therefore, the counter values are saved and reset to 0 after a fixed number of side exits has been taken. Figure 6 shows the basic formula for the computation of the similarity value. The input values are the current (cur) and previous ($prev$) values of a counter array with the length n. It is a slightly modified version of the *weighted* similarity computation introduced by Hind et al. [10]. The similarity value is a number between 0 and 1, where 0 means no similarity and 1 means perfect similarity. A value above a configurable but fixed threshold indicates a stable phase, a value below the threshold an unstable phase. The computation of the similarity value is simple and fast because only arithmetic operations and minimum computations are necessary. This is a positive side effect of the high granularity of traces: results can be

$$similarity = \frac{\sum_{i=1}^{n} \min\left(cur_i, prev_i\right)}{\min\left(\sum_{i=1}^{n} cur_i, \sum_{i=1}^{n} prev_i\right)}$$

Figure 6: Computation of phase similarity.

```
boolean checkStablePhase() {
  traceSim = computeSimilarity(traceCur, tracePrev);
  exitSim = computeSimilarity(exitCur, exitPrev);

  similarity = (traceSim + exitSim) / 2;
  return similarity > THRESHOLD;
}

double computeSimilarity(int[] cur, int[] prev) {
  for (i = 0; i < cur.length; i++) {
    sumMin += min(cur[i], prev[i]);
    sumCur += cur[i];
    sumPrev += prev[i];
  }
  return sumMin / min(sumCur, sumPrev);
}
```

Figure 7: Computation of phase similarity.

inferred directly from the counter values without the need for statistical correlation methods.

The trace counters and the side exit counters could be summed up together using this formula. However, these counters typically have a different characteristic: trace counters usually have much higher values because a loop is executed multiple times before it is exited at a side exit. To avoid that the trace counters dominate the similarity value, we compute the similarity separately for trace counters and side exit counters, i.e., we apply the formula twice. The overall similarity value is the arithmetic mean of the trace similarity and the side exit similarity.

Because of this, four arrays of counter values go into the computation formula: the current trace counters (*traceCur*), the current side exit counters (*exitCur*), the previous trace counters (*tracePrev*), and the previous side exit counters (*exitPrev*). Figure 7 shows the pseudocode for the computation formula. For each pair of a current and previous counter, the minimum is computed. If this value is big, both counters had a high value and so the execution behavior is similar. If only one of the counters was big, the minimum value is low and indicates a phase change. The sum of the minima is the divided by the minimum of the sums, which guarantees that the resulting value is between 0 and 1. If the arithmetic mean of the similarity values is above a certain threshold, the phase is stable.

3.3 Example

Figure 8(a) continues the example started in Figure 3. It shows the method `do()` of the Java Grande Series benchmark that actually calls the method `integrate()`. The method `integrate()` is called with three different values for the parameter `select`: first with the value 0 before the loop, and then with the values 1 and 2 alternatively inside the loop. In Figure 8(b), we split the loop into two loops so that the `select` values 1 and 2 are separated, which leads to a different phase behavior.

Figure 4 showed the trace tree when phase detection is not performed. The traces for all three values of `select` are contained in one trace tree. With phase detection, the trace tree is similar at first. The first call of the method `integrate()` with the `select` value 0 is too short to be detected as its own phase. Then the traces for the `select` values 1 and 2 are added to the trace tree. The loop is executed frequently enough so that the phase detection code is executed several

```
void do() {
  res[0][0] = integrate(0, 2, 1000, 0);

  for (int i = 1; i < rows; i++) {
    res[0][i] = integrate(0, 2, 1000, 1);
    res[1][i] = integrate(0, 2, 1000, 2);
  }
}
```

(a) Original version

```
void do() {
  res[0][0] = integrate(0, 2, 1000, 0);

  for (int i = 1; i < rows; i++) {
    res[0][i] = integrate(0, 2, 1000, 1);
  }
  for (int i = 1; i < rows; i++) {
    res[1][i] = integrate(0, 2, 1000, 2);
  }
}
```

(b) Modified version

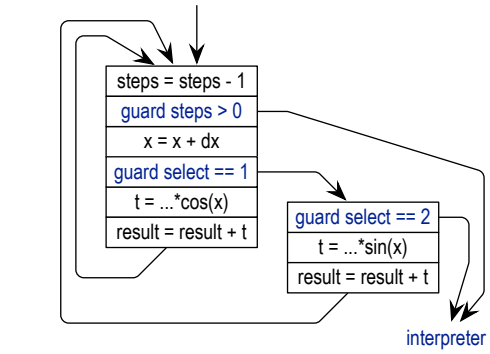
Figure 8: Java Grande Series example.

times. Because the method `integrate()` is no longer called with the `select` value 0, the trace counter for this path is 0. This means that the original trace tree is deleted and trace recording starts anew. This time, only the traces for the `select` values 1 and 2 are recorded. Figure 9(a) shows the resulting trace tree. This trace tree is smaller and requires fewer guard instructions to be executed because the comparison `select == 0` is no longer needed.

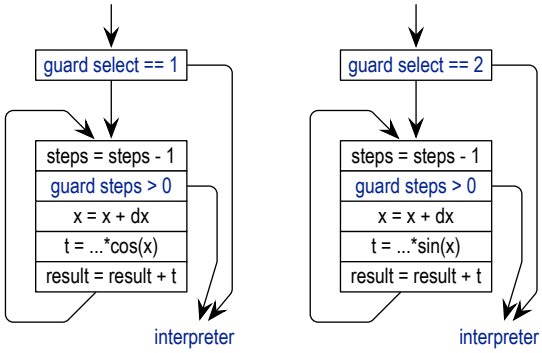
The modified source code of Figure 8(b) shows a different behavior. Because both loops are executed frequently, each loop is detected as a separate phase for the method `integrate()`. In the first phase, the only `select` value is 1, so there is only one trace in the trace tree. The guard with the comparison `select == 1` is now loop invariant and therefore moved to the loop header by the loop invariant code motion. Therefore, no check for the `select` value is performed inside the loop body, the machine code is specialized only for a single value. Figure 9(b) shows the resulting trace. When the first loop ends and the second loop with the `select` value 2 is executed, the guard fails and a side exit is taken to the interpreter. Phase detection then discovers that the trace is dead and recording starts anew. This time, the only `select` value is 2, so the machine code is specialized for this value, as shown in Figure 9(c). In summary, the modified example now has two stable phases, and the machine code is optimized for each phase separately.

4. PHASES BASED ON DATA FLOW

Trace compilation and phase detection complement each other well: trace counters are a good input data for phase detection, and phases can be easily used to guide optimizations of the trace compiler. A trace is recorded during an actual execution in the interpreter with specific values for all local variables and computations. It represents a class of equivalent executions, however the compiler can decide about the size of this equivalence class—it can cover all executions that follow the same control flow, or it can only cover executions with exactly the same values. The latter is clearly an overspecialization that is not useful in practice, but treating some rarely changing values as constants during



(a) Original version



(b) Modified, phase 1 (c) Modified, phase 2

Figure 9: Stable state traces for Series benchmark.

compilation and performing constant folding is a beneficial optimization. Phase detection allows all kinds of such specializations.

Traditionally, phases were defined only on control-flow properties such as profiles of program counters or conditional branches. However, the abstract problem definition and the computation of the similarity value is not specific to this. To specialize on certain data value properties in our system, it is only necessary to insert a guard instruction at the beginning of the trace. If the condition does no longer hold, a side exit is taken immediately after the trace execution has started. The counter for the side exit is incremented as usual. When the side exit is taken repeatedly, the algorithm presented in the previous section detects this as a phase change and handles it appropriately. Phase detection based on data flow extends the control-flow based approach, i.e., phases based on control flow are still detected even if no stable data values can be detected.

Any kind of specialization on concrete data values is supported by phase detection based on data flow. For scalar values, the code could be specialized for a certain range of values in order to eliminate, e.g., array bounds checks. For object values, the code could be specialized for objects of a certain type so that the compiler can eliminate type checks inside the loop. In our prototype implementation, we support only scalar variables that have a constant value because this information is easy to use for compiler optimizations: when it is known that a variable has always the same value, it can simply be replaced by a constant, and constant folding can optimize subsequent operations on this value. To

```
void daxpy(int n, double da, double dx[],
           int dx_off, double dy[], int dy_off) {
  ...
  for (int i = 0; i < n; i++) {
    dy[i + dy_off] += da * dx[i + dx_off];
  }
}
```

Figure 10: Java Grande LUFact example.

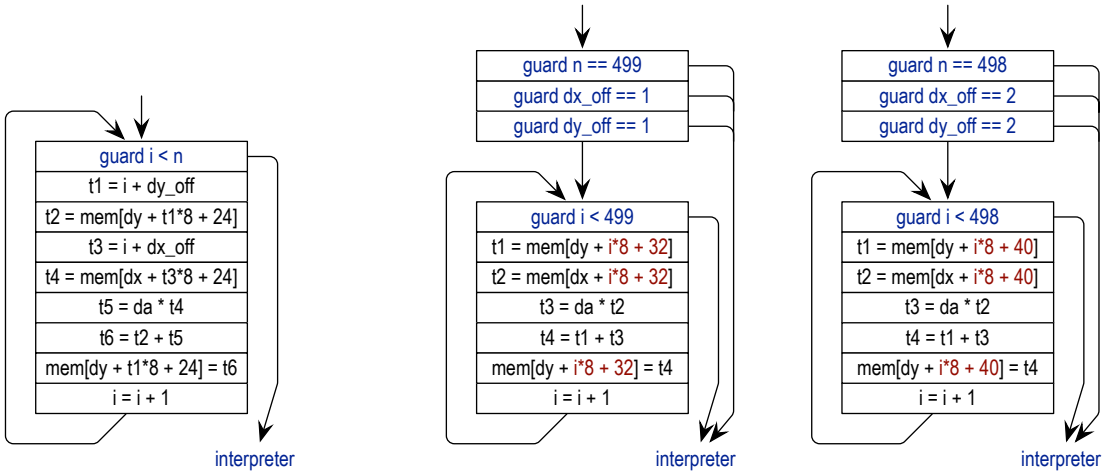
detect constant values, we instrument the interpreter to collect profile information at every backward branch, i.e., the interpreter compares the current values of the local variables with the values at the time of the previous backward branch and sets a flag whether the value is different or equal. Only local variables where the value was equal are then treated as constants during compilation. The profiling phase can be short and sketchy because phase detection corrects imperfect profiles with a low overhead.

4.1 Example

Figure 10 shows the most important computation loop of the Java Grande LUFact benchmark. The loop performs array accesses and arithmetic operations that involve mostly loop-invariant variables. They can be replaced by constants when it is known that the values are stable across many method invocations. Figure 11(a) shows the trace tree of the loop without phase detection. It consists of only one trace because there is no control flow inside the loop. The trace instructions are already low level instructions at a compiler state shortly before code generation, i.e., the array accesses have been lowered to memory accesses that use the indexed addressing mode. All trace instructions can be directly mapped to one machine instruction, only the guards require a compare and a conditional branch at the machine level.

The short profiling period during interpretation shows that the local variables `n`, `dx_off`, and `dy_off` have constant scalar values, while the values of `da` and `i` are changing. The variables `this`, `dx`, and `dy` are object references and therefore excluded by our analysis. The compiler assumes that `n`, `dx_off`, and `dy_off` are constant and replaces all uses with the appropriate constant values, which are 499, 1, and 1 in our case. It would be possible to replace `da` and even the initial value of `i`, which is changed inside the loop, with constants, but then the machine could not be re-used for even the next invocation of the method `daxpy()`, i.e., the phase would be extremely short. Figure 11(b) shows the resulting trace. Before the loop, guard instructions are inserted that trigger a side exit to the interpreter in case the assumption no longer holds. In the loop body, constant folding was able to eliminate two add instructions by folding them into the address arithmetic of the memory accesses. Additionally, the guard at the loop entry now has a constant operand.

After thousands of loop iterations, the method is called with different arguments, so the guards fail and side exits to the interpreter are taken. This leads to a rapid increase of the side exit counters while the trace counter drops to 0. The trace is deleted and the analysis starts anew. This time, different but still constant values are detected for the variables, so the trace is again specialized for the new values. Figure 11(c) shows the optimized trace for the second phase.



(a) Without data flow (b) With data flow, phase 1 (c) With data flow, phase 2

Figure 11: Traces when phase state is stable for Java Grande LUFact benchmark.

5. EVALUATION

We implemented trace-based phase detection for a research Java VM that is written entirely in Java itself. Our VM consists only of the interpreter and the JIT compiler. The VM runs on top of a host VM and delegates complex operations such as garbage collection and class loading to this host. This reduces the implementation effort and allows us to focus on compiler optimizations, while still supporting all parts of the Java VM specification. Writing the interpreter for Java in Java ensures not only type safety and platform independence, but also simplifies integration of the trace recorder and JIT compiler. However, some workarounds are necessary to invoke the machine code generated by the compiler directly from Java code, and it does not support multiple threads yet. While our system is a good playground for compiler optimizations, it has not yet been tuned for execution performance of large-scale applications.

All measurements were performed on an Intel Core 2 Duo processor T8300 with two cores at 2.4 GHz and 4 GByte main memory, running Mac OS X 10.5.4. The host Java VM was the Java 6 VM provided by Apple, i.e., the 64-bit Java HotSpot server VM 1.6.0_05. A large heap size was used to ensure that garbage collection does not affect the results. We use the Java Grande benchmark suite [3] for evaluation. It consists of several mathematical kernels of different code complexity. The computations are performed in loops, so Java Grande is well suited for our research VM. Our system traces only loops, however this is not a general limitation of trace compilation and trace-based phase detection.

5.1 Phase Statistics

This section presents statistics about the phases detected in the benchmarks. We compare phase detection using only control flow (as described in Section 3) and phase detection using data flow (as described in Section 4). For the baseline numbers, phase detection is disabled. All numbers of this section use a chunk size of 50, i.e., a check for a phase change occurs after 50 side exits have been taken.

Figure 12 shows the results for the benchmarks. The first three columns show general statistics of trace compilation. The overall number of trace trees, i.e., the number of hot loops that are detected, is usually quite low. On the one hand, the mathematical computations of the benchmarks are concentrated in few loops, on the other hand trace compilation effectively filters out unimportant code that surrounds the main computation loops. The maximum and the average number of traces per trees is also low. The most complex benchmark is MonteCarlo with 19 trees and a maximum of 11 traces per tree.

When phase detection based on control flow is enabled, the number of trace trees with phases is mostly equal to the overall number of trace trees. This is the expected result because a trace tree with one phase means that the behavior is stable for the entire time the loop is executed. Trace trees without a phase are executed long enough to get recorded, but too short for a phase to be detected. If a trace tree has more than one phase, a phase change is detected. The last but one column of this column group shows the number of trace trees that are deleted, i.e., the number of times a trace tree had unexecuted traces that were eliminated via phase detection. Finally, the last column shows the percentage of loop iterations that are executed in a stable phase. The remaining iterations are also spent in compiled code, but before the phase is stable or between stable phases. Loop iterations in the interpreter can be neglected because their number is low.

The Series benchmark shows the results that can be expected from the explanation in Section 3.3. We present the results for the original version and our modified version of the benchmark. The original version has only one stable phase. Before this phase is detected, the trace tree is deleted once to eliminate the unexecuted trace for the `select` value 0. The modified version has two stable phases, and the trace tree is deleted twice. Another benchmark with phase changes is the FFT benchmark, where 4 phases are detected for one method and 3 phases for another.

Phase detection based on data flow increases the number of detected phases for many benchmarks. Additionally, constant local variables are also discovered for trace trees that still have only one phase. The maximum number of constant local variables per trace tree is 5 for a tree of the FFT benchmark. This tree has 28 phases with varying val-

	no phase detection			phases using control flow					phases using data flow						
	trees	traces per tree		trees w/	phases per tree		deleted	% iter.	trees w/	phases per tree		deleted	% iter.	const locals	
		max	avg.	phases	max	avg.	trees	in phase	phases	max	avg.	trees	in phase	max	avg.
Series	3	3	1.67	1	1	1.00	1	99.1	1	1	1.00	1	99.5	1	1.00
Series (modified)	3	3	1.67	1	2	2.00	2	98.5	1	2	2.00	2	98.8	2	2.00
LUFact	4	4	2.00	4	1	1.00	0	99.4	4	384	96.75	383	71.2	3	2.97
HeapSort	3	2	1.33	3	1	1.00	0	89.9	3	2	1.33	1	98.9	1	0.25
SOR	3	3	2.33	3	1	1.00	0	99.7	3	1	1.00	0	99.7	4	2.00
Crypt	3	2	1.33	3	1	1.00	0	97.4	3	1	1.00	0	97.4	0	0.00
FFT	8	4	1.75	8	4	1.63	5	97.3	8	28	5.13	47	90.1	5	3.46
SparseMatmult	4	2	1.25	3	1	1.00	0	99.3	3	1	1.00	0	99.3	1	0.67
MonteCarlo	19	11	1.74	19	1	1.00	0	99.1	19	3	1.11	4	99.1	4	0.67
RayTracer	2	5	3.00	2	1	1.00	0	99.9	2	1	1.00	0	99.9	0	0.00

Figure 12: Phase detection statistics for Java Grande benchmarks.

ues of these variables. Because of the increased number of phase changes, the percentage of loop iterations that are in a phase is lower compared to control-flow based phase detection. The trace tree of the benchmark LUFact that was used as the example in Section 4.1 shows the highest number of phases. Three local variables are replaced by constants that change in each phase. The exact number of phases depends on the chunk size. The theoretical maximum would be 500 phases because the method is called with 500 different parameters, but this would require a small chunk size that is not useful in practice. Our default configuration detects 383 phases that are long enough to be relevant in practice.

Figure 13 shows the details of the phase behavior for this trace tree. The x-axis of the diagrams show the profile time, i.e., a number that is increased at every side exit. The similarity values for phase detection that are computed at these points are shown on the y-axis. The similarity value is always between 0 and 1, where 0 means no similarity and 1 means perfect similarity. The part of the similarity graph in Figure 13(a), which shows the first phases of the trace tree, is highly regular: at first, a new phase is detected after about 500 side exits. However, each new phase is slightly shorter than the previous.

Figure 13(b) shows a part of the trace tree where the phases are already much shorter. They have a length of only about 100 side exits. When the length drops below a threshold, the phase is too short to be detected as stable. The local variables change before their constant value can be identified. This leads to a final phase where no specialization based on data flow is performed. This phase is then stable for the remaining time of the benchmark run.

5.2 Impact of Chunk Size

The most important configuration parameter of phase detection is the chunk size. A large chunk size leads to infrequent checks for phase changes and therefore reduces the impact of outliers and noise in the application behavior. However, it also leads to a long response time on phase changes, i.e., a phase change is detected later than with a small chunk size. Short phases can also not be detected with a large chunk size. The chunk size should therefore be as small as possible. In our system, a chunk size of 50 showed good results. Figure 14 shows the impact when the chunk size is further reduced for one trace tree of the HeapSort benchmark.

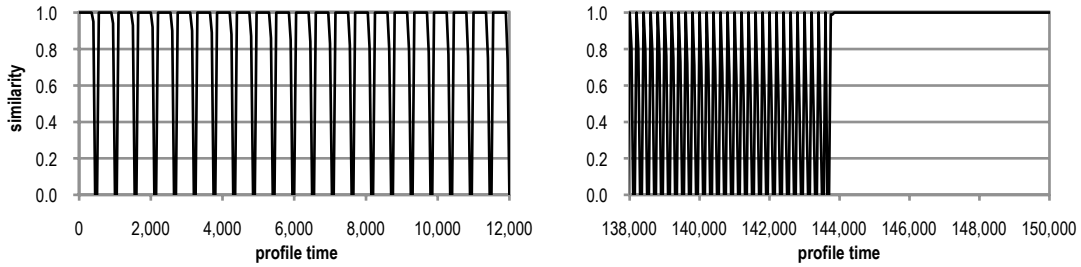
The corresponding trace tree has about 10 different side exits that are taken, so it has a reasonably complex control flow. We show the similarity values for a chunk size of 10, 20, and 50. With a chunk size of 10, the similarity value is quite noisy and frequently drops below 0.5, which we consider the threshold for a phase change. This would lead to a high number of phases although the application behavior does not change. With a chunk size of 20, the similarity value is above 0.6, and with a chunk size of 50 there are no more outliers that lead to false phase changes. The response time is still good because an actual phase change is detected after 50 invocations of the trace tree in the new phase.

6. RELATED WORK

Hind et al. provide an abstract definition of phase detection (called phase shift detection in their work) [10]. When the two functions *granularity* and *similarity* are defined, an input string can be split into phases. They then provide suitable definitions for granularity and similarity that lead to a well-behaving system where certain properties like having a unique solution can be proven. Our phase detection algorithm and our similarity formula follow these sound definitions. They evaluate their system using a profile of all conditional branches collected from various Java benchmarks. Because the profile spans the whole application run, they detect global phases, while our system detects phases locally per trace.

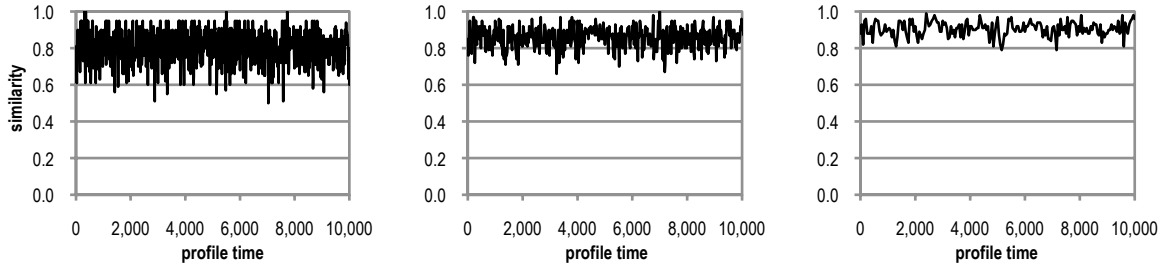
Nagpurkar et al. present a framework for online phase detection algorithms, define a large class of online phase detectors, and evaluate their accuracy [14]. The baseline for the accuracy comparison is an offline profile analysis, considering loops and repeated method invocations at the source code level. They use several Java benchmarks to compare the impact when different parameters of the phase detection algorithms are modified, like the window policy and the model policy. Similar to Hind et al. the input data for phase detection are profiles of conditional branches, and they consider only global application phases.

Das et al. use local phase detection to perform region monitoring [4]. They discovered that their earlier approach using global phase detection did not yield information that



(a) First phase changes (b) Last phase changes and final stable phase

Figure 13: Similarity values for the LUFact benchmark, method `Linpack.daxpy()`.



(a) Chunk size 10 (b) Chunk size 20 (c) Chunk size 50

Figure 14: Similarity values for the HeapSort benchmark, method `NumericSortTest.NumSift()`.

was useful for dynamic optimization systems.Therefore, they treat each code region individually. Although they mention traces as one definition of a local region, they do not use traces themselves. They form regions based on the value of the program counter at a certain sampling interval, which requires statistical correlation coefficients. In contrast, our formulas are much easier because our counters are already tied to a certain trace, i.e., to a range of program counters. They evaluate their approach using benchmarks from the SPEC CPU2000 benchmark suite, but do not describe an actual use of the phase information.

Gu et al. perform phase detection using hardware performance counters that track L1 cache misses [9]. They state that the cache miss rate correlates well with the method execution behavior. The phase information is then used to schedule method recompilation at different optimization levels in a Java VM. This approach is similar to our use of the phase information, however they operate at the granularity of methods.

Shen et al. predict global phases using an offline profiling phase and then transform the program using binary rewriting [15]. Phases start at inserted marker points, and the first few executions of a phase are used to predict all later executions. They explore the repeated access of variables (memory cells) for phase behavior, but they do not look at the actual values of the variables.

Sherwood et al. propose the use of *Basic Block Vectors* (BBVs) to capture and detect phase behavior [16]. BBVs are used for counting the frequency of basic block execution for a given interval. They perform a basic block distribution analysis to find small portions of the program that represent the behavior of the full program. Phase information is used to find preferred simulation points in the application in order to achieve a representative sample of its execution. In [17] they define a unified profile architecture that can capture, classify, and predict next phase changes. Their phase tracker can be used to trigger hardware or software optimization.

Dhodapkar at al. use instruction working set signatures to detect phase changes [5]. Phase changes are detected by comparing consecutive signatures, using a metric called the *relative signature distance*. When the relative signature distance exceeds a certain threshold, a phase change is detected. Dynamic microarchitecture reconfiguration is invoked in response to the phase changes. In [6] they present a comparison of three phase detection techniques based on *basic block vectors*, *instruction working sets* and *conditional branch counts*.

Kistler et al. present a system for continuous program optimization [13]. They profile the application and use this data to schedule recompilations with different compiler optimizations. If the profiling data changes between two consecutive time stamps, a change in the application behavior is detected. Although not called phase change in the paper, the definition is similar.

Gal et al. introduce trace-based compilation and trace trees [7, 8]. They integrate trace recording and trace compilation into the JamVM, a lightweight Java VM for embedded devices, as well as into the TraceMonkey JavaScript VM. Before utilizing traces for compilation, they were used in the binary optimization system Dynamo [2]. In this system, machine code is optimized at run time, which differs from compilation of Java bytecodes because machine code misses all of the high-level information that bytecodes provide.

In contrast to traditional compilation at the granularity of methods, traces cover only frequently executed loops which start and end in the middle of methods. This can be seen as an advanced combination of *on-stack replacement* (OSR) and *deoptimization*. OSR [12] allows the transition to machine code for long running loops whose execution started in the interpreter. Deoptimization [11] stops the execution of

machine code in the middle of a method so that execution continues in the interpreter. However, both are expensive operations that are not intended for frequent use, as it would be required for phase change detection.

7. CONCLUSIONS

We presented a novel method for local phase detection that is based on trace compilation. Counters are incremented at the backward branch of traces and when traces are exited at side exits. When the distribution of the counter values in the current period is significantly different from the previous period, a phase change is detected. We use the information about stable phases to guide compiler optimizations. Because the counter values of traces are already at a high granularity, more complex definitions of a phase are possible than in other approaches. This allows us to define phases based on data flow, e.g., to begin a new phase when the value of a local variable changes. The evaluation showed that our approach identifies phases for several benchmarks.

Acknowledgements

Parts of this effort have been sponsored by the National Science Foundation (NSF) under grants CNS-0615443 and CNS-0627747, as well as by the California MICRO Program and the industrial sponsor Sun Microsystems under Project No. 07-127. Further support has come from generous unrestricted gifts from Google and from Mozilla, for which the authors are immensely grateful.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and should not be interpreted as necessarily representing the official views, policies, or endorsements, either expressed or implied, of the NSF, any other agency of the U.S. Government, or any of the companies mentioned above.

8. REFERENCES

[1] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12. ACM Press, 2000.

[3] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[4] A. Das, J. Lu, and W.-C. Hsu. Region monitoring for local phase detection in dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 124–134. IEEE Computer Society, 2006.

[5] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the International Symposium on Computer Architecture*, pages 233–244. IEEE Computer Society, 2002.

[6] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 217–227. IEEE Computer Society, 2003.

[7] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–478. ACM Press, 2009.

[8] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 144–153. ACM Press, 2006.

[9] D. Gu and C. Verbrugge. Phase-based adaptive recompilation in a JVM. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 24–34. ACM Press, 2008.

[10] M. J. Hind, V. T. Rajan, and P. F. Sweeney. Phase shift detection: A problem classification. Technical Report RC-22887 (W0308-006), IBM Research Division, 2003.

[11] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[12] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336. ACM Press, 1994.

[13] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, 2003.

[14] P. Nagpurkar, M. Hind, C. Krintz, P. F. Sweeney, and V. Rajan. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–123. IEEE Computer Society, 2006.

[15] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 165–176. ACM Press, 2004.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57. ACM Press, 2002.

[17] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the International Symposium on Computer Architecture*, pages 336–349. ACM Press, 2003.