Phases in Branch Targets of Java Programs

Technical Report CU-CS-983-04

Matthias Hauswirth
Computer Science
University of Colorado
Boulder, CO 80309
hauswirt@cs.colorado.edu

Amer Diwan
Computer Science
University of Colorado
Boulder, CO 80309
diwan@cs.colorado.edu

ABSTRACT

Recent work on phase detection indicates that programs behave differently at different points in their execution. This paper looks at phases in more detail with respect to indirect calls, which are common in object-oriented programs. We divide phases into three categories: code dependent, context dependent, and data dependent. Of these, code-dependent phases are the easiest to exploit and require no special hardware or compiler support. Data-dependent phases, on the other hand, do require significant hardware or compiler support. Fortunately, we find that at least for indirect calls, data-dependent phases are rare for a collection of Java benchmarks.

Keywords

phase behavior, branch target prediction, feedback-directed optimization

1. INTRODUCTION

Recent studies on phase detection [9, 10, 11, 7, 6, 2, 3] indicate that programs behave differently at different points in their execution. For example, consider a long-running server application that handles a continuous sequence of requests. Some of the requests may require intense computation while others may require many memory or disk accesses. There are two ways of achieving the best performance from these applications: (i) The underlying hardware mechanisms can adapt to the changing behavior of the application. For example, the hardware branch predictor observes the sequence of executed branches and their outcome and can therefore adapt its predictions based on the behavior that it observes. (ii) A feedback-directed optimizer may observe program behavior using low-overhead sampling and reoptimize the code when the behavior changes [1]. Both approaches have their strengths and weaknesses. Hardware approaches are generally more efficient than software approaches (e.g., code reoptimization is expensive) but software approaches know more about global program behavior than hardware.

Technical Report CU-CS-983-04
University of Colorado at Boulder
December 2004

The relative strengths and weaknesses of the hardware and software approaches suggest that for rapidly changing program behavior, hardware approaches, and more specifically speculation mechanisms, will probably be most suitable. On the other hand, for behavior that changes more slowly, software approaches, and more specifically feedback-directed optimizations, may be more suitable since there is more time between behavior changes to recover the cost of the reoptimization. To better understand this tradeoff, this paper explores how the behavior of method invocations changes over time during program runs.

To conduct this study, we compiled and ran a number of Java programs using a modified version of the Jikes RVM. Our modifications to the Jikes RVM allow us to get detailed information about both low and high level events during an application run. To increase the generality of our results, we present data for not just the commonly-used SPECjvm98 benchmark suite but also for four more realistic applications.

Our results reveal that as far as method invocations are concerned, few of them exhibit any change in behavior: i.e., most method invocations are monomorphic and call the same method every time. This was a surprise to us: we expected Java programs, and particularly the real Java programs in our test suite, to exhibit significant polymorphism. Since few of the method invocations are actually polymorphic, a hardware or software mechanism designed to exploit varying behavior over time should be tightly focused on the few invocations that actually need the mechanisms.

The remainder of this paper is organized as follows. Section 2 introduces the background of our work. Section 3 describes our infrastructure and benchmarks. Section 4 analyzes our results. Sections 5 and 6 describe related and directions for future work. Finally Section 7 concludes.

2. BACKGROUND

A *program phase* is a period of time over which some aspect of program behavior (e.g. the call targets of virtual method invocations) is relatively stable. We classify program phases into three classes:

Code-dependent phases . Loops and function calls in programs lead to the repetitive execution of the same code. If a block of code behaves the same whenever it is executed but different blocks may have different behavior, we get *code-dependent* phases.

Optimizing for code dependent phases is trivial: each block of code can be optimized based on the aggregate profile (over the complete program run).

Since many speculation mechanisms index their information using the PC value, even the simplest mechanisms (e.g., last value instead of ones that use more context or history information) can easily handle code-dependent phases.

Context-dependent phases . If the behavior of a block of code depends on its calling context (e.g., its immediate caller) it yields *context-dependent* phases.

Software approaches can exploit context-dependent phases by using some form of specialization. Specialization effectively turns context-dependent phases into code-dependent phases.

In order to recognize and exploit context-dependent phases in hardware, the hardware needs to keep track of some form of context information (e.g., correlated branch prediction instead of last-direction-taken).

Data-dependent phases .

If the behavior of a block depends on data values it leads to *data-dependent phases*.

Software approaches to exploit data-dependent phases need feedback-directed optimizations. In other words, the compiler must reoptimize code while it is running based on its recent history.

We do not know of any hardware designed to specifically exploit data-dependent phases.

The above classes are not mutually exclusive. Many phases in a realistic program may fit more than just one class. The above classification helps to focus on phases which need to be detected dynamically. For feedback-directed reoptimization, dynamic phase detection might be required because phases of a given class can not be delineated statically (for data-dependent phases), or because the collection of an offline profile is too costly (context-dependent phases). Since data-dependent phases are the most difficult to exploit, the rest of the paper focuses on on them.

Phase Indicators

To study when the same code exhibits different behavior due to different data we focus on indicators which signal differences in data and are potentially exploitable by optimizations.

The following list shows examples of possible indicators and how they might signal a phase transition:

Indirect call targets . An indirect call site that has been calling one target for a while starts to call another target.

Return addresses . A return instruction that has returned to a specific caller for a while suddenly returns to a different caller.

Conditional branch outcomes . A conditional branch that has been going one way for a while starts to go the other way.

Load values . A load instruction that has been loading values with a certain property starts to load values with another property.

Load addresses . A load instruction that has been loading from a certain set of addresses starts to load from another set of addresses.

3. APPROACH

We now describe how we perform our measurements and our benchmark programs.

Infrastructure

We use our Vertical Profiler for the Jikes Research Virtual Machine (JikesRVM) to analyze the phase behavior of a set of Java benchmark programs. Our Vertical Profiler is an extension to JikesRVM 2.2.0 on Linux/Intel. Vertical profiling denotes the concept of profiling low-level events (like indirect calls) and relating them to high-level causes (like polymorphic method invocations).

JikesRVM is a virtual machine that runs Java programs by compiling them to native code at runtime. It comes with two Java bytecode to native code compilers. The fast baseline compiler does not perform any optimizations. The optimizing compiler performs a complete set of optimizations, most importantly inlining and register allocation.

JikesRVM, including its compilers and memory managers, is mostly written in Java. Only wrappers around system calls, and the loader, is written in C. Thus, our results include the entire execution of the system (except for the small amount of code written in C), including garbage collection and compilation.

We use the BaseBaseMarkSweep configuration of JikesRVM. This means that all the code is baseline compiled[1], and that the configuration uses a mark and sweep garbage collector.

Instrumentation

In this paper we focus on one phase indicator: indirect call targets. This indicator is particularly interesting since virtual method invocations, which cause significant overhead in object-oriented programs, are usually compiled to indirect calls.

Our Vertical Profiler places an instrumentation right before each indirect call. The instrumentation writes a unique program point ID, identifying the instruction, and the target address of the call into the trace buffer.

We analyze the trace buffer at every buffer overflow. We summarize the events found in the trace buffer, and write out that summary at the end of the run.

The online analysis for indirect calls keeps a list, for each call site, of all target addresses with corresponding edge counts. It also simulates a branch target predictor and counts the mispredictions for each call site over time.

Additionally, our Vertical Profiler gathers structural information about the program. This includes descriptions of all the types, the methods of each class, the units of code that are assembled, and the program points that are instrumented.

Our infrastructure can collect data for other phase indicators (such as branch outcomes) as well. We hope to report on the outcome of using other phase indicators in the future.

Benchmarks

Table 1 presents our extensive benchmark suite which consists of a few micro benchmarks, the SPECjvm98 suite, SPECjbb2000, and an additional set of more complex Java

[1]We expect opt-compiled code to have fewer monomorphic method calls, because the optimizing compiler can use inlining at monomorpic call sites. We plan to quantify this difference in future work.

programs.

We use at least the size 100 inputs for the SPECjvm98 benchmarks. For some of them we chose even bigger and more interesting inputs, with the goal of provoking the phase behavior we expect to find in server applications.

The additional benchmarks are more realistic applications. They are either server-centric or have an object-oriented design. We chose large and complex inputs to reflect the inputs a real server application might see.

Benchmark	Description
Empty	One class with empty main method. To measure VM overhead.
PolyPhased	One call site with phased polymorphic method invocations.
PolyPhaseless	One call site with phaseless polymorphic method invocations.
Compress	Compresses and decompresses five different 1...3MB files. Repeats this five times.
Jess	Java expert system shell. Solves the wordgames.clp number puzzle problem.
Raytrace	Raytracer, rendering a dinosaur scene into a 600 by 200 color bitmap.
Db	Runs a sequence of queries (mostly sorts) in memory on an 1MB database.
Javac	Java compiler, compiling JavaLex.java four times in a row.
Mpegaudio	MPEG audio decoder decoding a 3.2MB mp3 file.
Mtrt	A multithreaded version of raytrace, rendering the same scene with 4 threads in parallel.
Jbb	Multithreaded server application, predominantly simulating the middle tier of a 3-tier application.
Xalan	XSLT processor, applying different XSL stylesheet to different XML files.
Pdom	Persistent DOM database, running a sequence of XPath-like queries on various Pdom database files.
Soot	Object-oriented Java bytecode optimizer.
SableCC	Highly object-oriented compiler compiler.

Table 1: Description of benchmarks

Benchmark	# of Classes	# of Methods	Assembly Unit		# of Ind. Calls
			Base	RT	
Bootimage	380	6394	6394	41	38900
Empty	230	130	130	291	1340
PolyPhased	235	139	139	301	1370
PolyPhaseless	235	139	139	301	1370
Compress	252	198	198	436	2481
Jess	398	605	605	1134	5446
Raytrace	265	315	315	627	3892
Db	247	201	201	457	2733
Javac	411	937	937	4433	9933
Mpegaudio	292	357	357	767	5113
Mtrt	265	317	317	627	3897
Jbb	448	1014	1014	3113	11272
Xalan	877	2091	2091	13473	20857
Pdom	378	597	597	1621	4329
Soot	969	1145	1145	3748	17506
Sablecc	613	1523	1523	6940	13452

Table 2: Static structure of benchmarks

Table 2 shows the structure of those benchmarks. For each benchmark we show the number of classes, methods, assembly units, and instrumented program points. An assembly unit is a chunk of machine code produced by the assembler. An assembly unit often corresponds to the code of a compiled method. There are different clients of JikesRVM's assembler: the baseline compiler (Base) and the optimizing compiler are the most important ones. There are other clients, all of them part of the JikesRVM runtime system (RT). Of those, the lazy compilation trampoline generator (which generates tiny stubs for dynamically loaded methods) is the most prevalent one. We also show the number of instrumented program points (indirect calls).

The first line in Table 2 shows the structure of the bootimage. Since the bootimage is part of every benchmark, and always looks exactly the same, we factored it out into a separate line. The subsequent lines only contain information about individual benchmark programs. For individual programs we only show data for classes and methods that are actually used in our runs. To get the complete information for a benchmark, one has to add the *Bootimage* and the respective benchmark's lines.

The *Empty* micro benchmark shows the startup overhead of the virtual machine. The application consists of only one class, with just an empty `main()` method. Thus, when comparing two benchmarks, one can subtract the numbers of *Empty* from the numbers of the respective benchmark, to get the numbers that are specific to that benchmark. Since the startup behavior depends on the context (command line arguments, heap sizes, whether benchmark classes are loaded from files or from JAR archives) to some degree, we did not factor out the startup costs shown in *Empty* from the other benchmarks.

We use the *PolyPhased* and *PolyPhaseless* microbenchmarks as a bounds and sanity check in our analysis of indirect calls. *PolyPhased* consists of one frequently executed virtual method invocation site and exhibits four distinct phases. Each phase uses a different call target, but during a phase the target is constant. Thus we can clearly observe four phases, each with different behavior. *PolyPhaseless* is similar in that it also has one important virtual method invocation site with four distinct targets. But it exhibits no phase behavior since successive invocations always have different targets.

4. RESULTS

While most method invocations in Java have the potential to be polymorphic, only a few of them may exhibit interesting phases. Broadly speaking, there are three kinds of polymorphic method invocations; only the last kind (Phased polymorphism) is amenable to techniques that exploit phases.

Polymorphism for modularity Polymorphic method invocations may exist in a program to facilitate future extensions. Even though these method invocations have the potential to be polymorphic, they will always call the same target at run time. The literature contains many techniques (e.g., class hierarchy analysis or preexistence-based inlining) for removing the inefficiencies associated with these method invocations.

Phaseless polymorphism The second form of polymorphism actually leads to indirect call sites that invoke multiple targets at runtime. We call it phaseless because invocations to the different targets are heavily interleaved, such that no longer phase of consecutive invocations of the same target method does appear.

We created the *PolyPhaseless* micro benchmark to demonstrate this behavior. PolyPhaseless has a call site with four different target methods (plus four lazy compilation trampolines). The call site is inside a loop, and the target method in every iteration is different from the previous one.

Phased polymorphism The last form of polymorphism appears, when there are clear phases, and in each phase there is a clear bias toward a different call target.

We created the *PolyPhased* micro benchmark to exhibit this behavior. PolyPhased is similar to PolyPhaseless except that it has four distinct and equal phases in each of which the target method of the virtual method call is the same.

Of the three categories described above, phased polymorphism and polymorphism for modularity can be exploited with software and hardware mechanisms. Phaseless polymorphism, on the other hand, is difficult to optimize for both software and hardware approaches.

Low-level Behavior

Data-dependent phase behavior is only visible in call sites with more than one target. The number of targets observed at a call site at runtime is called its target arity. We call sites with arity 1 *runtime-monomorphic*, and sites with arity >1 *runtime-polymorphic*.

Table 3 shows the percentage of calls executed at call sites with the given target arities. We can see that the overwhelming majority of calls happens from runtime-monomorphic call sites. From the real benchmarks, only *Compress*, *Mpegaudio* and *Xalan* have more than 10% runtime-polymorphic call executions. This significantly limits the benefit that can be gained by reoptimizing a block of code due to a runtime-polymorphic call site changing its bias towards a different target.

To focus our search for call sites amenable to phase-based optimizations, we measure the miss rates of two theoretical branch target predictors. A branch target predictor [5] predicts where a branch instruction is going to jump to. We look at the miss rates of a *bias predictor* and a *last value (LV) predictor*. The bias predictor statically predicts an indirect call instruction to always have the biased target (the target with the highest probability determined by an offline profiling run). The LV predictor predicts the call site always to have the same target it had the last time it was executed. The size of our LV predictor is unlimited, which means that it keeps track of the last target of every call site.

Table 4 shows the percentage of runtime-polymorphic call executions. It also lists the percentage of indirect call executions that are mispredicted by the bias predictor. Indirect calls that have a low bias miss rate are uninteresting with respect to phase behavior because a low-bias miss rate means that one of the targets is called most of the time. Finally the table shows the percentage of call executions that miss in the LV predictor. High LV predictor miss rates are an indication of constantly changing behavior. Since the behavior most amenable to optimization is a heterogeneous sequence of phases, where each phase has internally homogeneous behavior, we are most interested in situations with high bias miss rates (heterogeneous sequence) and low LV miss rates (homogeneous phases).

To see how the bias and LV miss rates can help to identify programs with phases, let's examine the data for the PolyPhased and PolyPhaseless benchmarks. We see that PolyPhaseless has both a high bias and LV miss rate indicating that it does not have distinct phases, at least at the level of indirect calls. On the other hand, PolyPhased has a high bias miss rate but a low LV miss rate indicating that that it has distinct phases with respect to indirect calls.

In Table 4 we can see a considerable discrepancy between the indirect call executions with arity >1 and the indirect call executions missing in the bias predictor. The bias predictor miss rate is often very close to 0. The reason for this is simple: JikesRVM's compilers create indirect calls for all method invocations. The target of the call is usually the body of the method we want to invoke. But since Java supports lazy compilation (compilation of methods when they are invoked for the first time), JikesRVM often uses a small block of code, the *lazy compilation trampoline*, as a stub for the future method. Thus the first invocation of that method will lead to calling the trampoline. Subsequent calls will call the target method. This leads to many call sites that are runtime-polymorphic, even though the corresponding Java method invocations actually are monomorphic. And because there is just one call to the trampoline, but many calls to the real target method, the bias miss rate of that call site will be almost 0.

Temporal Behavior

We manually inspected the temporal behavior of indirect call sites that have at least two call targets that are not trampolines. We particularly focused our search on sites with a high bias miss to LV miss ratio (sites with potentially distinct, homogeneous phases).

The most interesting call site we found lies in method `TreeMap.rbInsert()` in the *Sablecc* benchmark. This call site makes up about 1.2% of the overall 10.3 billion executed indirect calls. It has a bias miss rate of 33.56% and an LV miss rate of only 0.01%. It is an invocation of method `Comparable.compareTo()`. Figure 1 shows the behavior of that call site over time. The X axis shows the time in intervals. An interval consists of 10 million indirect call executions. The Y axis shows the number of times this call site invokes each of its five different targets per interval.

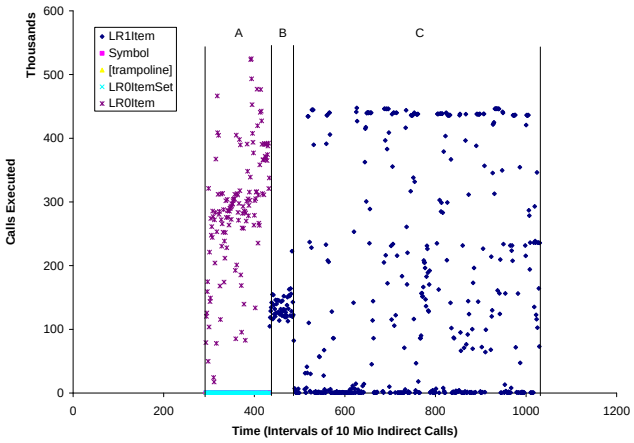

Figure 1: Call targets over time

One of the call targets is a lazy compilation trampoline, which is invoked only once (in interval 293). Two of the targets (in class `LR1Item` and `LR0Item`) are called millions of times. And the remaining two targets (`Symbol` and `LR0ItemSet`) are only called a few thousand times. We can see that during the first 292 intervals the call site is never executed. Then, in phase *A* from interval 293 to 433, we can see many

Benchmark	Executions	1	2	3	4	5	6	7	8	9	>9
Empty	17,222,051	98.45%	0.36%	0.02%	0.35%	0.30%	0.01%	0.00%	0.00%	0.51%	0.00%
PolyPhased	84,803,459	20.54%	0.08%	0.00%	0.07%	0.06%	0.00%	0.00%	79.13%	0.10%	0.00%
PolyPhaseless	84,816,757	20.56%	0.08%	0.01%	0.07%	0.06%	0.00%	0.00%	79.12%	0.10%	0.00%
Compress	606,605,590	54.91%	45.04%	0.00%	0.01%	0.01%	0.00%	0.00%	0.00%	0.01%	0.00%
Jess	1,030,430,602	98.23%	1.29%	0.36%	0.02%	0.01%	0.01%	0.00%	0.00%	0.01%	0.07%
Raytrace	667,736,259	97.62%	1.80%	0.05%	0.31%	0.14%	0.01%	0.07%	0.00%	0.01%	0.00%
Db	1,042,054,699	96.66%	3.31%	0.00%	0.00%	0.02%	0.00%	0.00%	0.00%	0.01%	0.00%
Javac	1,046,457,653	97.28%	1.33%	0.30%	0.12%	0.12%	0.07%	0.05%	0.03%	0.14%	0.54%
Mpegaudio	209,065,752	78.85%	17.86%	0.01%	3.06%	0.08%	0.08%	0.00%	0.01%	0.04%	0.00%
Mtrt	779,416,908	94.02%	5.35%	0.04%	0.27%	0.12%	0.01%	0.19%	0.00%	0.01%	0.00%
Jbb	3,823,274,352	96.47%	3.42%	0.02%	0.00%	0.06%	0.01%	0.00%	0.00%	0.00%	0.01%
Xalan	12,810,724,214	83.65%	7.94%	6.29%	0.55%	0.01%	1.05%	0.00%	0.00%	0.51%	0.00%
Pdom	12,107,961,857	96.08%	3.77%	0.10%	0.00%	0.03%	0.02%	0.00%	0.00%	0.00%	0.00%
Soot	261,580,829	97.86%	0.72%	0.03%	0.90%	0.23%	0.01%	0.00%	0.04%	0.11%	0.10%
Sablecc	10,335,741,475	96.89%	1.78%	0.00%	0.02%	1.20%	0.01%	0.09%	0.00%	0.00%	0.00%

Table 3: Percentage of indirect call executions, by call target arity

Benchmark	Executions	Arity >1 Executions	Bias Misses	LV Misses
Empty	17,222,051	1.55%	0.32%	0.42%
PolyPhased	84,803,459	79.46%	59.42%	0.09%
PolyPhaseless	84,816,757	79.44%	59.41%	79.21%
Compress	606,605,590	45.09%	0.02%	0.01%
Jess	1,030,430,602	1.77%	0.32%	0.31%
Raytrace	667,736,259	2.38%	0.03%	0.02%
Db	1,042,054,699	3.34%	0.01%	0.01%
Javac	1,046,457,653	2.72%	0.43%	0.34%
Mpegaudio	209,065,752	21.15%	1.49%	0.06%
Mtrt	779,416,908	5.98%	0.03%	0.02%
Jbb	3,823,274,352	3.53%	0.02%	0.01%
Xalan	12,810,724,214	16.35%	0.57%	1.03%
Pdom	12,107,961,857	3.92%	0.04%	0.04%
Soot	261,580,829	2.14%	0.16%	0.13%
Sablecc	10,335,741,475	3.11%	0.46%	0.01%

Table 4: Percentage of indirect call executions missing in predictors

calls to `LR0Item` (on average 288,000 per interval), together with the few calls to `Symbol` (average 403) and `LR0ItemSet` (average 47). Phase *B*, from interval 434 to 486, consists of an average of 134,000 invocations of `LR1Item` per interval (standard deviation: 18,400). Finally phase *C*, from 487 to 1030, still calls `LR1Item`, but on average 155,000 times per interval (standard deviation: 177,000).

A static optimizer using an aggregate profile of this call site would have decided to optimize for calling `LR1Item`. This would have left us with 33.56% suboptimal calls at this site. Using feedback-directed reoptimization, we could decide to initially optimize for calling `LR0Item`, and then reoptimize the code in interval 434, so it is optimal for calling `LR1Item`. In this ideal situation (where detecting the phase transition, identifying the bias target for the new phase, and reoptimizing have no cost) we would have ended up with only 0.05% suboptimal calls.

A hardware speculation approach using the simple LV branch target predictor would only have caused 0.01% suboptimal calls. But the additional information available to a feedback-directed optimization system would allow the optimizer to optimize away the calling overhead completely, and to specialize the inlined code for a further performance gain.

5. RELATED WORK

Barnes et al. [2] sketch a general approach to detect phases in a managed runtime environment. In their description they focus on conditional branch probabilities as a phase indicator. They do not distinguish between the different classes of phases we propose. In a different paper [3] they classify conditional branches based on their phase behavior. Their "Multi High" category consists of branches that have a clearly different bias during different phases. Using a subset of the SPEC CPU95 and SPEC CPU2000 benchmarks they find that up to 3% of static branches exhibit this behavior.

The Basic Block Vectors (BBV) in [9, 10, 11] are a valuable tool for phase identification. But even though a BBV indirectly captures context-dependent and data-dependent phases (a change in the indicator for data-dependent phases, like an indirect call target, leads to successive differences in executed basic blocks), their direct focus is on executed code and thus on code-dependent phases. A successor paper [11] presents an implementation to capture an approximation of the BBV in hardware, and to classify and predict phase based program behavior with this metric. Applying their approach to measuring frequent value locality they find that overall the top 16 load values only represent 10% of loads, whereas by using the top 16 values in each phase they can capture almost 50% of the executed loads.

Dynamic working set analysis [7] uses a bit vector to track which basic blocks were touched. It is an abstraction of BBV, since the only information in the working set signature is whether a block was executed or not (but not how many times).

An application running in a virtual machine environment requires considerable runtime support. Two of the key runtime services provided by modern virtual machines are the garbage collector and the just in time compiler. Hu and John [8] look at the data cache performance of those two subsys-

tems compared to the actual application code. They use the notion of JVM phase to denote a computation performed by such a subsystem. They also show time-dependent cache miss rates for two of their benchmarks, but do not correlate the time-varying cache performance to higher level causes.

6. FUTURE WORK

We are currently studying different phase indicators, like conditional branches, return addresses, load values, and load addresses. We are interested in quantifying the opportunities for reoptimization based on those indicators.

We also would like to investigate the effects of copying garbage collection on data-dependent phase behavior by replacing the mark and sweep collector with a copying collector. Since code could be moved around, call targets would change after every garbage collection.

Furthermore we are looking at the influence of JikesRVM's optimizing compiler on data-dependent phase behavior.

Multi-threading is an important factor in the manifestation of phases during program execution. If a program is multithreaded, then different computations get interleaved in time, and thus different phases are mingled together. It would be interesting to investigate the phase behavior of programs on a per-thread basis.

7. CONCLUSIONS

On the software level a runtime optimizer can exploit larger grained phases by reoptimizing code at phase transitions. And on the architecture level speculation hardware can be provided with information to adjust to finer grained phases.

A system that reoptimizes code based on profiling at run time incurs a considerable run-time cost. Blindly reoptimizing code is going to waste precious time. Our results indicate that the opportunities for reoptimization are sparsely distributed over the code. Thus an efficient reoptimization system benefits from not just focusing its profiling and reoptimization efforts on frequently executed code, but from additionally focusing on the subset of code that exhibits data-dependent phase behavior. This can not only reduce the profiling overhead, but it can also lead to better targeted and thus more beneficial optimizations.

Being able to detect phase changes at low cost also enables run-time system support for hardware speculation. Previous work [4] shows that compiler support can improve load value prediction. By exploiting knowledge about phases in a program, we expect to be able to improve the hints the compiler and runtime system can contribute to confidence estimators of value predictors or branch target predictors.

8. ACKNOWLEDGMENTS

We would like to thank the Jalapeno/JikesRVM group at the IBM T. J. Watson Research Center for developing and maintaining the great open source Java virtual machine we used for gathering our traces.

9. REFERENCES

[1] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 111–129. ACM Press, 2002.

[2] R. D. Barnes, E. M. Nystrom, M. T. Conte, and W. mei W. Hwu. Phase profiling in a managed code environment. In *First Workshop on Managed Run Time Environment Workloads*, 2003.

[3] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. mei W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *International Symposium on Microarchitecture (MICRO)*, pages 233–244, 2002.

[4] M. Burtscher, A. Diwan, and M. Hauswirth. Static load classification for improving the value predictability of data-cache misses. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 222–233. ACM Press, 2002.

[5] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. In *International Symposium on Computer Architecture (ISCA)*, pages 274–283. ACM Press, 1997.

[6] S. Clarke, E. Feigin, W. C. Yuan, and M. D. Smith. Phased behavior and its impact on program optimization. Unpublished submission to FDO, 2002.

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

[8] S. Hu and L. K. John. Comparison of jvm phases on data cache performance. In *First Workshop on Managed Run Time Environment Workloads*, 2003.

[9] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 2001.

[10] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[11] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *International Symposium on Computer Architecture (ISCA)*, 2003.