Experiences in Designing a Robust and Scalable Interpreter Profiling Framework

Ian Gartley   Marius Pirvu   Vijay Sundaresan   Nikola Grcevski
IBM Canada Ltd.
{igartley,mpirvu,vijaysun,nikolag}@ca.ibm.com

Abstract

Profile directed feedback (PDF) is a well known technique used to drive many compiler optimizations like basic block ordering and guarded devirtualization. These optimizations are particularly crucial in order to achieve good throughput performance in JEE applications that have a large code footprint. To effectively apply optimizations that rely on profiling information, a just-in-time (JIT) compiler must have access to profiling information that is accurate. One common source of profiling information in a Java virtual machine (JVM) is the interpreter. Typically methods are interpreted as a program ramps up, during which profiling information can be collected. However, obtaining useful and accurate information for large enterprise-class applications can be a challenge because of the memory and performance overhead associated with collecting and processing the large volumes of profiling data that is generated. This paper describes the challenges in maintaining the balance between throughput performance and profiling overhead in a production JIT compiler that is used by the IBM JDK. The scope of the performance overhead in terms of throughput, memory footprint and startup speed for large JEE class applications is introduced and various engineering solutions that were tried are detailed and compared in terms of experimental results. We found that the throughput improvement due to interpreter profiling (IP) can be as high as 58%, whereas the overhead measured in terms of application startup time could cost up to 57%. Our solutions to reducing profiling overhead managed to reduce the startup cost to only a few percent while maintaining the full throughput benefit. By discussing these approaches, this paper offers a balanced and practical overview on how to make PDF work well for enterprise-class applications in a production JIT compiler.

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE...$15.00

1. Introduction

One of the main advantages of a just-in-time (JIT) compiler in a Java virtual machine (JVM) is that it can collect and utilize profiling information without involving the user in any way. This is possible because the JVM manages both the collection as well as resulting optimization automatically by employing various heuristics and analyses. Profile directed feedback (PDF) is an extremely important technique for improving performance of a broad spectrum of programs. Small programs with relatively few hot methods as well as larger programs with very flat execution profiles can benefit to a large extent.

Typically, the following information is useful for a JVM to collect:

- Sampling information about relative execution frequency of different methods.
- Control flow graph (CFG) edge frequency information, which can be used to derive basic block frequencies; knowledge about program hot spots can be used in optimizations like block ordering, loop unrolling, and so on.
- Call receiver type information, which can be used to perform guarded devirtualization.
- Value profiling of other useful expressions, such as:
  - The number of copied bytes at a given System.arraycopy which can be used to generate optimal platform specific sequences.
  - Type information at checkcast/instanceof which can be used to optimize type checks.
  - Loop bounds which can be used to specialize a loop for a particular input value.
- Profiling information which measures how often some unexpected paths are taken, for example - contention at synchronization points.

One approach to profiling is to instrument the JIT generated code, run the profiling body for a short period of time while collecting data, and use the data at a later point in a subsequent recompilation. This approach has been widely studied and improved upon in literature, as described in [1–3] and others. In our terminology this is referred to as JIT profiling and IBM's J9 JVM implements a version derived from the work of Arnold and Ryder [1]. Many small and medium sized Java programs (including most typical Java benchmarks) can benefit from JIT profiling. Such programs are quite sensitive to path length and depend heavily on traditional optimizations involving loops, arrays, inlining and type/constant propagation. By recompiling a small subset of key methods and aggressively optimizing them based on profiling information, performance can be improved substantially.

However, there exists a distinct class of applications that pose unique challenges: enterprise-class applications. Applications, such as IBM's Websphere Application Server product and the JEE applications that it can host, can be very complex programs involving tens of thousands of classes and methods that are more or less equally important. They tend to have a very large instruction footprint with code that is very call and branch intensive, which puts significant pressure on the cache hierarchy of the machine. Therefore, profile driven compiler optimizations such as basic block ordering and guarded devirtualization/inlining are extremely important for improving throughput. Unfortunately, the sheer volume of methods that must be instrumented and recompiled due to the flat execution profile makes JIT profiling prohibitively expensive. An alternative is to collect profiling data while the methods run interpreted. This Interpreter Profiling (IP) approach relies on the fact that interpretation is already a slow process, thus its costs are more tolerable (in a relative way). Moreover, recompilation overhead completely vanishes, as methods that contribute to performance are compiled anyway. In contrast to JIT profiling, the literature is very scarce when it comes to IP. In fact, despite our best efforts we could not find a relevant publication that gives this topic the coverage it deserves.

Regardless of where profiling is done, a significant proportion of the bytecodes in any program must be profiled. This inevitably leads to a need to collect and reduce a large amount of profiling data, while keeping the performance and footprint overhead within an acceptable threshold. Therefore, it becomes a necessity to implement a scalable infrastructure to cope with the high volume of data that gets generated. Reducing the collected data to a form that is consumable by optimizations within the compiler is an equally challenging task.

The main contribution of this paper is a comprehensive overview of the issues we encountered while implementing a robust and scalable IP framework. Given the lack of a definitive piece of related work on this subject, and the critical importance of implementing an effective profiling framework in dynamic runtimes in general, this paper plays an important role in showing how beneficial such frameworks can be, as well as how careful engineering can make the difference between being able to afford such a framework or not. This paper focuses on illustrating performance results in enterprise-class Java applications; despite their importance, such applications have not received much attention in the literature mainly because they require complex and demanding runtime environments that are harder to setup and properly configure. However, for completeness, we also examine several other typical Java benchmarks.

Section 2 is an overview of the basic IP infrastructure employed in the J9 VM. Section 3 describes the experimental methodology used to evaluate various ideas to reduce profiling overhead. Section 4 introduces the extent of the overhead from a performance as well as footprint standpoint that is observed with the basic implementation of IP. Section 5 details each of the techniques we employed to reduce the profiling overhead and presents experimental results showing their effectiveness. Finally, Section 6 discusses related work in this area, and how our paper differs from the prior research into effectively balancing profiling overhead with throughput performance.

2. Basic infrastructure of interpreter profiler

In J9 VM the interpreter profiling infrastructure maintains a loose coupling between the interpreter and the JIT compiler, separating the responsibility of each component. The interpreter collects the data, but the JIT compiler reduces and uses the information. The interpreter maintains a buffer in each thread of execution that is used to store the profiling data. The data in the buffer consists of a series of profiling records containing the program counter (bytecode PC) and profiling data. The format of the profiling data is specific to the bytecode referred to by the PC. After a thread's buffer fills up, the interpreter reports a "buffer full event", which is listened to by the JIT compiler. When the JIT compiler receives the "buffer full event", it starts parsing the data and stores the information in a synchronized global hash table for later reuse when compiling the program code. The following relevant pseudocode is shown:

```
While not EOF(buffer)
   PC = read (buffer, word size);
   OpCode = getOpCodeType (PC);
   DataSize = getOpCodeDataSize (OpCode)
   Data = read (buffer, DataSize);
   addOrUpdateProfileData(PC, Data);
End While
```

It is important to understand the difference in format between what is stored in the thread's buffer and the synchronized hash table: the per-thread buffer contains the data as it is written out by the interpreter while executing each bytecode. Therefore, if a particular bytecode PC was executed multiple times, there would be multiple records written out to the executing thread's buffer. The synchronized hash table is not thread-specific and contains the consolidated results from processing all the per-thread buffers. In particular there is at most a single entry in the hash table for a given bytecode PC, which is also used as the key into the hash table. The unique entry for a given bytecode PC contains the total number of times that bytecode was executed, as well as the number of times each kind of event/value was encountered (for example, how many times a branch was taken, or how many times the receiver was of a particular type at a virtual call). The format of different records in the per-thread buffers varies by bytecode type and this also results in a different format for the entries in the hash table. There are reasonable defaults for how many distinct events/values are tracked at each type of bytecode. After the JIT compiler parses the buffered profiling data, the per-thread buffer is reset and the interpreter starts filling it with fresh data.

Using per-thread profiling buffers has several advantages over other kinds of profiling approaches. Because the profiling is done on the application thread and the buffer data structures are allocated for each thread, writing data to the buffer does not require any synchronization. The parsing process, on the JIT compiler side, is only synchronized at the point when data is stored in the global hash table. With the buffered profiling approach, the process of collecting and storing the profiling data is completely independent of each other. The separation of the two processes solves the memory footprint issue to some extent because the JIT compiler can decide when to discard the profiling data that was collected. For example, when a program segment is fully optimized, the profiling data for that code can be disposed. The memory footprint introduced by the interpreter for each per-thread buffer is constant. Another important advantage of collecting IP data in per-thread buffers is that, a trace of the application execution is contained in the buffer. Namely, because a profiling sample is taken for each executed call bytecode while interpreting the code, call path information is automatically generated and available to the JIT compiler (at least when considering only interpreted methods). As can be imagined, dynamic call path information is of great value to the method inlining compiler optimization. Given that call path information exists for a given group of methods, the inlining optimization can make informative decisions on which call subgraphs to inline and how deep.

During compilation of a method, the JIT compiler queries the profiling data on-demand. For example, when the compiler makes a decision about how to layout the code given an "if" statement, the compiler queries the profiling data for that PC. If profiling data for the PC exists in the hash table, the profiling framework returns the branch taken and not-taken counts so that the compiler can make an optimal decision about how to order the code.

Additional support is also required for code unloading. The J9 VM has a cooperative threading model: all application threads have to yield at "GC safe points" for the garbage collector (GC) (and consequently class unloading) to proceed. Because class unloading only occurs when all application threads are paused, there is no concern about application threads changing interpreter profiling state concurrently. If a class is unloaded, the JIT compiler must discard any data in its hash table (in Java a class can be unloaded after there are no references to its class loader object). Any profiling data related to that class must also be removed from any per-thread buffers. Class unloading is a more common occurrence than might be expected in certain kinds of JEE applications where auto generated classes and short lived class loaders are used in large numbers. Therefore the interaction between class unloading and interpreter profiling is an important problem that must be dealt with separately. For the per-thread buffers the approach we take is a somewhat conservative one in that all the information in the per-thread buffers of each application thread is discarded. For the hash table the stale data is not proactively purged when a class gets unloaded. Instead, when an entry is accessed, a check is performed to decide if it originated from a class that has been unloaded. This is done by comparing the bytecode PC against a set of bytecode PC ranges that are known to correspond to unloaded classes. Another issue that can occur is that a buffer entry refers to an unloaded class (for example, receiver type of a call). This case is handled in the compile-time query that accesses value profiling information by comparing the profiled value against a set of class address ranges that are known to correspond to unloaded classes.

3. Experimental methodology

3.1 Description of benchmarks

From the performance point of view, both high peak throughput and short startup time are desireable. These two goals are usually antagonistic - improving one of them often leads to a degradation of the other. The situation is no different with interpreter profiler: collecting more profiling information leads to better code and higher throughput, but negatively affects startup time (and vice-versa). For this reason we will present experiments showing both these metrics.

To evaluate the benefits and overhead of IP we use one enterprise-class application Apache DayTrader 2.0 and a few benchmarks from the SPECjvm2008 suite.

Apache DayTrader [4] is a JEE application that runs on top of an application server and simulates an online trading system. DayTrader leverages many of the Java EE technologies that are very popular today, like Java Servlets, JavaServer Pages (JSPs), Java database connectivity (JDBC), Java Message Service (JMS), Enterprise JavaBeans (EJBs) and message-driven beans (MDBs). As such, it exercises both the Web container and the EJB container of the application server. Following a three tier architecture design, DayTrader also requires a client driver application to apply load as well as a database backend for data persistence. In our case the client application (called JIBE) was developed in-house, the application server that hosts DayTrader is the Websphere Application Server v8 64-bit and DB2 v9.7 plays the role of the data-base backend. In conjunction with DayTrader throughput, we will also measure Websphere Application Server startup time which is a key product goal for the IBM JVM.

SPECjvm2008 [5] is a benchmark suite that measures the performance of a JVM. It stresses the processor and memory subsystems of a machine and, in contrast to DayTrader, has no network activity and very little file I/O activity. We picked this suite because it is somewhat more recent, but mainly because it includes startup benchmarks. Due to space constraints and to reduce clutter we only show a few benchmarks from the suite (compiler.compile,

	Classes loaded	Comp. methods	#exec. methods	sys/user cpu ratio
DayTrader	18.8K	10K	58K	0.22
compiler.compile	2.6K	3.1K	10K	0
serial	2.2K	1K	8.1K	0
sunflow	2.2K	1K	8.2K	0
xml.transform	3.8K	4.4K	15.3K	0.05
scimark.fft.large	2.1K	0.5K	7.4K	0

Table 1. Characteristics of benchmarks.

serial, sunflow, xml.transform, scimark.fft.large) together with their startup counterparts (startup.compiler.compile, startup.serial, startup.sunflow, startup.xml.transform, startup.scimark.fft.large). Because we do not strictly follow SPEC reporting rules in our experiments, throughout this paper we refer to the subset of the SPEC suite chosen by us as jvm08.

Table 1 presents a few key characteristics of the selected benchmarks. `Comp.methods` represents the number of methods compiled in a run with default JVM parameters, while `#exec.methods` represents the total number of methods executed. The table wants to point out the fact that DayTrader is a much more complex benchmark with a bigger footprint and that it stresses all aspects of a machine. The smallest application in terms of code is scimark.fft.large and as such we expect it to be influenced the least by interpreter profiling (be it either throughput or startup time).

3.2 Experimental platform and methodology

The experimental platform is an x86 server with two Intel® Xeon® X5670 (Westmere EP) processors having 6 cores each, clocked at 2.93 GHz and running SUSE Linux. If a database backend is required (for DayTrader) a second machine of similar characteristics hosts an instance of DB2 v9.7. The two machines are connected directly to each other by 1 Gbit crossover Ethernet cable. The client application that is required for DayTrader runs on the same machine as the DB2 database for convenience.

The JVM used is Service Refresh 1 (SR1) of the IBM Java Development Kit (JDK) release for Java Version 7. We employed the 64-bit version of this JDK and compressed references were enabled.

The Websphere Application Server startup time is measured from the moment the command is launched till the moment the Websphere Application Server emits the *open for e-business* message. Starting the application server also means starting the applications that are installed on top of it (in this case DayTrader). After Websphere Application Server starts up, the client generates various HTTP requests to it in a series of one minute bursts interspersed with 10 second delays. To allow the application server to warm-up we ignore the throughput results for the first 5 bursts and compute the average of the last 3 bursts.

For jvm08 we used the following benchmark options:

```
-ikv -ict -bt 2 -i 1 -wt 30 -it 60
```

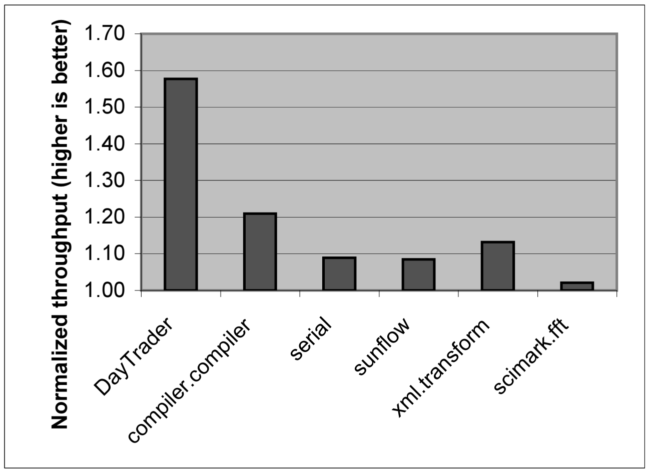

Figure 1. Effect of interpreter profiling on throughput. Results are normalized to NoIP.

All experiments were repeated four times and the results were averaged. JIT profiling was always turned on (this is the default behavior in IBM's J9 VM).

4. Effect of interpreter profiling on runtime performance

Figure 1 illustrates the effect of IP on application performance. The peak throughput improves for all benchmarks and in some cases dramatically (DayTrader). This demonstrates that both big and small applications can be helped by information provided by IP. Smaller applications (such as scimark.fft.large) will likely benefit less because the performance is determined by a few key methods and the JIT profiler can successfully provide the same information as IP. For large applications however, it is simply not feasible to instrument thousands of methods.

Unfortunately, the benefits of IP do not come for free. Its overhead is mostly felt during the incipient stages of applications and thus startup (and rampup) might be affected. Figure 2 shows the effect of the base implementation of IP on startup time. Again the effect is more pronounced on Websphere Application Server/DayTrader (57%) due to the large number of methods that run interpreted and the time required to compile them. Given the attention and scrutiny Websphere Application Server startup time receives from the customers, such levels of regressions are not acceptable from the product standpoint.

There are three sources of overhead that stem from IP operation: (1) interpreter collecting relevant data in the profiling buffers; (2) JIT parsing the profiling buffer content; (3) JIT updating the global hash table, or querying it during compilation. To give the reader an idea of the contribution of each source, in Figure 3 we show Websphere Application Server startup times when sources are added one by one. Results are normalized to the case where interpreter profiling is not used (which receives a score of 1). In the

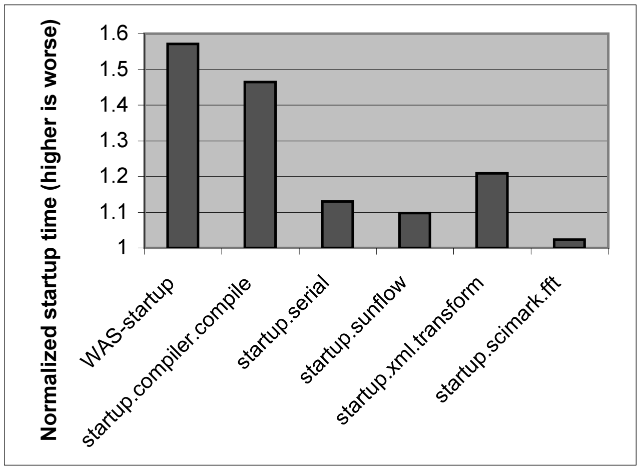

Figure 2. Effect of interpreter profiling on startup time of applications. Results are normalized to NoIP case.

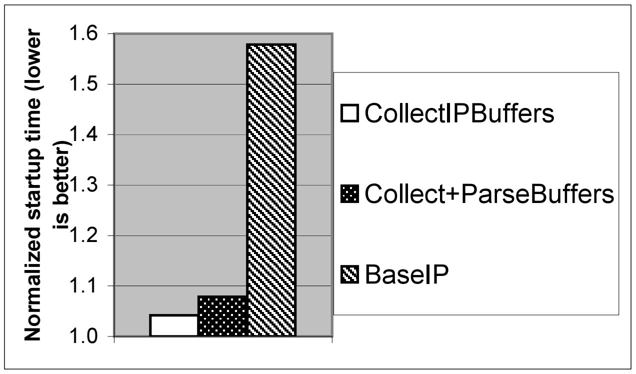

Figure 3. Experiments showing different sources of IP overhead (Websphere Application Server startup). Results are normalized to NoIP case.

`CollectIPBuffers` case we let the VM collect IP information in the per-thread buffers, but discard such buffers when they reach the JIT. The overhead of collecting is about 4%. In the `Collect+ParseIPBuffers` case we go one step further and allow the JIT to parse the information into records, but stop short when updating the global hash table. This operation adds another 4% overhead. Finally, the last case represents the base interpreter profiling mechanism that introduces most of the overhead. In the next section we describe some techniques that we used to address each of these overhead components.

To give an insight on why some of the heuristics were chosen we present here some statistics related to the data collected by the interpreter profiler. Currently, IP collects information about branch, invoke (call), checkcast/instanceof and switch bytecodes. Table 2 shows the distribution of these types of records found in profiling buffers. The takeaway is that the most common type of record is the branch. This insight suggested that it might be possible to skip the processing of some branches without losing important information and implicitly without degrading throughput.

	branch	invoke	chkcast	switch
DayTrader	69.1m	50.7m	5.7m	1.2m
compiler.compile	16.0m	13.5m	2.2m	0.5m
serial	7.8m	2.7m	0.3m	0.1m
sunflow	7.9m	2.9m	0.3m	0.1m
xml.transform	49.2m	32.3m	2.5m	1.4m
scimark.fft.large	5.8m	1.5m	0.2m	0.1m

Table 2. Distribution of IP records found in IP buffers.

	branch	invoke/chkcast	switch
DayTrader	91.3k	168.3k	1.1k
compiler.compile	13.6k	22.1k	0.2k
serial	9.7k	12.9k	0.1k
sunflow	10.0k	13.4k	0.1k
xml.transform	24.6k	39.2k	0.6k
scimark.fft.large	9.0k	12.0k	0.1k

Table 3. Distibution of IP entries found in IP hash table.

Table 3 presents another aspect of IP data, specifically the distribution of different types of entries found in the global IP hash table. First, it should be observed that the most common type of entry is the one used for invokes and checkcast operations (the same type of entry is used for both). Second, the number of entries is much lower than the number of processed records which implies that many bytecodes are executed multiple times.

Both Table 2 and Table 3 show that the amount of IP data that is processed and stored in DayTrader is much larger than in any other analyzed application. In contrast, the smallest amount of data is generated by scimark.fft.large. This is in line with the observation that Websphere Application Server/DayTrader startup is the most affected by interpreter profiler while startup of scimark.fft.large is affected the least.

We performed some experiments to determine which type of profiled bytecodes influences DayTrader throughput the most. By disabling the data collection for each type of bytecode in turn we found that branch profiling information is worth around 25.7% and invokes slightly more (26.2%); checkcasts come a distant third (5%), while switches have virtually no effect (0.2%).

5. Techniques to reduce IP overhead

5.1 Description of implemented techniques

As shown, the basic implementation of the interpreter profiler has a significant amount of overhead and quickly becomes one of the main startup performance bottlenecks. In this section we discuss the techniques that are used to reduce interpreter profiling overhead.

a) Remove synchronization from the hash table containing the parsed interpreter profiling information. The global hash table is implemented as an array of buckets, with each bucket being a single linked list of entries (chained hash table). Because it can be accessed by several threads in parallel there are concerns about synchronization issues. The general solution for synchronized access is locking (mutual exclusion), but, from the performance point of view this brings three disadvantages: (1) scalability is affected; (2) path length is increased and (3) under the covers locking uses atomic machine instructions that are expensive. In the literature there are a few examples of algorithms that offer lock-free (non-blocking) access to hash tables [6–9]. Unfortunately, they require the usage of some sort of atomic operation (for example, compare-and-swap). Because we target many architectures and platforms we strive to write portable code and avoid assembly language as much as possible. Another disadvantage is (arguably) that a sophisticated solution might introduce intermittent, hard to catch software bugs. Nevertheless, as a preliminary evaluation of such lock-free algorithms we built a solution based on a very lightweight synchronization framework implemented in assembly. When a thread tries to access the hash table but finds it in use, it immediately aborts the operation. This way there is no overhead due to contention and the synchronization is as fast as it can be. The experimental results were not satisfactory in the sense that the overhead was hardly reduced. This was a sign that the bulk of the overhead comes from the atomic instructions themselves and a lock-free algorithm based on CAS instructions might not bring much relief.

Hence, we decided to pursue an algorithm that carefully inserts entries at the front of the linked list for a bucket without any synchronization. The lack of synchronization has four implications:

1. To maintain correctness and avoid segmentation faults deletions from the hash table are not permitted
2. When two threads try at the same time to add an entry to the same bucket one of them might get lost. Losing one entry is not going to create functional problems or affect the usefulness of the profiling data. The likelihood of such an event is very low not only because the window of opportunity for the race condition is very narrow, but also because most applications use a low number of threads during startup when most of the interpreting activity takes place
3. For platforms that have a weak memory consistency model and do not guarantee the order of stores (System P) we must insert a memory barrier between the instructions that set various fields in a newly created entry and the instruction that links the entry to the bucket (which should be placed last)
4. Hash table resizing is not permitted

Entries that must be deleted (maybe because the bytecodes they refer to have been unloaded) are marked with a ”deleted bit” rather than actually removed from the hash table, and the flag gets queried before accessing the entry. Once again there is no functional correctness implication of updating the ”deleted bit” in an unsynchronized manner because each thread would do necessary checking anyway before using the entry (that is the ”deleted bit” should be viewed as a quick test). Note that it is possible to actually delete entries with the ”deleted bit” set at program points where it is known that another thread is not concurrently modifying the hash table.

While the lack of synchronization does not allow hash table resizing we can still do it on GC points when application threads are paused. The downside is that the GC pauses sometimes increase which is not desirable. As a palliative we chose a default hash table size that accommodates most applications. For bigger applications we experimented with an alternative technique: when the load factor of the hash table reaches 1, we create a secondary, much larger hash table. New entries will be added to the secondary table, but searches have to lookup both tables. Our experiments revealed that the performance advantage offered by this scheme was minuscule and that it did not warrant the extra complexity and memory footprint. Moreover, some of the techniques that are detailed next reduced the number of entries in the hash table and obviated the need of a secondary hash table.

b) Reduce class unloading overhead. The main overhead due to class unloading comes from having to check if a given hash table entry is for a bytecode PC from a method that has been subsequently unloaded, especially when the number of unloaded methods is large. Our idea to address this problem was to keep a global ID for class unloading and increment it every time the GC unloads classes. Each hash table entry is also tagged with its own class unload ID and this field is set to be equal to the global ID for class unloading every time it is validated as an entry that does not correspond to a bytecode PC for an unloaded method. Therefore, if the class unload ID for the entry equals the global ID for class unloading, we can avoid checking the validity of the entry because we can assume that no class unloading happened since the last time the check was performed. Because the global ID for class unloading is written by the GC thread when all application threads are paused (class unloading happens during a GC), there is no race condition on that global variable.

c) Heuristics for starting/stopping interpreter profiling. Some applications (including the Websphere Application Server in certain configurations) keep interpreting during the whole application execution period due to reflection code which constantly generates and disposes classes and methods. In such cases the interpreter profiling overhead is noticeable (1-2%) even at steady state. Note that the nature of reflection code is such (frequent native calls and so on.) that performance typically does not depend on profiling.

To alleviate this problem the JIT contains a mechanism to completely turn off the interpreter profiling after the application start-up and warm-up phase has ended: The JIT monitors the number of samples that fall on interpreted and compiled method bodies. When the density of interpreted samples drops under a certain threshold and this behavior lasts for a while, the JIT assumes that the bulk of important methods has already been compiled and there is no additional benefit to be had from profiling any further. Thus, the JIT informs the interpreter to stop populating IP buffers.

The benchmarks analyzed in this paper do not exhibit this continuous interpretation problem, so this heuristic to stop IP will not show a tangible benefit (thus, this heuristic is not included in our experimental results). However, in other applications, by stopping profiling the JIT can regain the steady state performance it had to give up because the JVM was still interpreting and collecting profiling information.

If IP is turned off completely, a backup mechanism is required to turn it on later in case a significant phase change in application behavior occurs. Such a phase change would only matter from a profiling viewpoint if new classes are introduced that require IP to collect information about the new code. To deal with such scenarios the JVM uses the class loading rate to detect that the application is starting to load new code (classes) and therefore must start collecting profiling information before the methods get compiled. In short, if class loading rate increases above a certain threshold the JIT turns IP on again.

Some applications might have phase changes, but all the required classes are loaded at startup; therefore, monitoring the class loading rate will fail to detect the phase change. To cope with such scenarios the JIT looks to see if the density of interpreted samples increases above a threshold and turns interpreter profiling on appropriately.

In some extreme cases the dynamic generation of new classes/methods is done at such a scale that many samples fall on interpreted methods causing a restart of the interpreter profiler. To safeguard against this behavior an additional heuristic is in place: the more classes are unloaded the more IP records are skipped from being added to the hash table. The mechanism is tuned to be rather conservative as to only catch extreme cases. Again, none of the benchmarks presented in this paper exhibit this type of behavior, so we do not show experimental results of this heuristic.

d) Heuristics to skip some profiling records. We have observed that branches are encountered more frequently than other types of bytecodes in typical programs (see Table 2). The amount of interpreter profiling information written to the per-thread buffers is directly proportional to the number of times the interpreter executes these bytecodes; therefore it is reasonable to expect that the buffers would have more information related to branches.

On the other hand, looking at the distribution of entries present in the hash table we observed that branches have dropped to second place. If we divide the number of records by the number of entries for each category of bytecodes we find that a branch is producing 700-2000 IP records (depending on the benchmark), while an invoke/checkcast produces 100-700 IP records. This is a clue that it may be feasible to skip branch records to reduce overhead without losing throughput.

We used a simple heuristic to reduce the overhead of processing buffers based around the assumption that skipping information for branches (as long as done in a random manner) would not seriously impact the quality of the information or the outcome of JIT optimizations. We use an algorithm based on a random number generator to decide whether the JIT should add a given branch profiling record into the global hash table. Note that this technique is not expected to reduce the footprint of the global hash table significantly because it is unlikely that we can omit creating a hash table entry entirely due to skipping some profiling samples. More likely, the JIT would have a smaller execution frequency in some of the hash table entries.

e) Asynchronously process interpreter profiling buffers on a dedicated thread. In our base implementation of IP, each application thread has a profiling buffer which is filled while methods run, and when the buffer becomes full, the application thread calls a JIT compiler hook and starts to process the information from the buffer. While doing so, the application thread is not making progress in terms of executing Java bytecodes and so we can term this approach to be synchronous in terms of the processing it does.

One of the most significant changes we made to the IP infrastructure was to introduce a separate pool of threads dedicated to the task of processing profiling buffers. There are multiple benefits to this approach that make the significant increase in complexity of the profiling infrastructure worthwhile. Apart from better utilization of excess CPUs on a multiprocessor machine, the other benefit is that the profiling buffer processing can follow an asynchronous model. In other words, application threads can add their full profiling buffers to a queue to be processed and then continue executing Java bytecodes while one of the threads processes the buffer.

This change introduces several new possibilities in terms of the heuristics that can be applied. Tuning must be done to choose a reasonable default for the thread pool size as well as adapting to the changing size of the queue of IP buffers. We found that having a single dedicated thread is good enough in practice, and we also found that dropping buffers when the size of the queue of buffers exceeds a certain threshold is also beneficial in reducing the overhead. Another interesting heuristic involves decision making around when to fall back to using application threads to process the profiling buffers. In fact it is possible that a certain number of application and profiling threads are working on processing the data at the same time. This is particularly relevant when the volume of profiling data is so large that the profiling threads are unable to keep up and so application threads have to be used to ensure the JIT does not lose too much valuable profiling information.

The orchestration done between application and profiling threads also deserves some description. When the profiling buffer gets full the application thread calls a JIT compiler hook that acquires a monitor, adds the profiling buffer to the working queue and sends a signal to the profiling thread. The profiling thread then grabs a free buffer from a pool, attaches it to the working thread and then, it releases the monitor. The profiling thread usually sleeps while waiting for some work to arrive. When it receives a signal from an application thread, it dequeues the first profiling buffer from the working queue and starts processing it (no monitor is held while processing the buffer). After this, the buffer is placed back into the pool of buffers that are available for use by application threads. If there is more work to be done, the next buffer from the working queue is processed; otherwise, the thread goes to sleep. The JIT compiler keeps a counter with the number of outstanding buffers to be processed on the queue. When this number exceeds a certain threshold, the application thread can either process the profiling buffer itself or it might choose to discard the contents of the buffer. The profiling thread is run at normal (same as the application threads) priority to minimize the interactions with other threads in the JVM.

f) Profile only the last N invocations of the interpreted method before it gets compiled. The motivation here is to prevent the interpreter from generating information unless there is a very good chance that the method will be compiled, and to limit the generation of samples so that we have just the amount of data required to make the appropriate compile time decisions. Apart from avoiding excessive information from methods that would eventually get compiled, this heuristic is also effective at avoiding information from methods that do not get compiled at all (for example, avoid profiling some very short lived auto-generated bytecodes). This avoiding of "endless" profiling is more significant than it might seem at first, because there is a class of applications that rely on run-time generation of bytecodes to perform tasks and in such (admittedly distinctive) applications, interpreter profiling can impact the steady state performance by up to 50%.

Another advantage is that the profiling information is biased towards the program's behavior just before the method was compiled which might be more representative of steady state behavior by virtue of omitting samples from the really early startup phase. The interpreter also avoids writing profiling information to the buffer if the method has been queued for compilation or already compiled (possible if the interpreter is still executing a long running loop in the method).

Experiments and tuning are paramount for arriving at the best value for N (number of invocations) to profile, and in J9 VM we chose a value of 1000. We found that we were able to minimize the throughput impact while reducing overhead significantly by using this technique. Moreover, the number of entries and the footprint of the hash table dropped by more than 50% (the reduction is attributed to not collecting IP samples for methods that are invoked very infrequently).

5.2 Evaluation of implemented techniques

Figure 4 illustrates the effect of the techniques described in the previous section on start-up time. Their order on the chart corresponds to the order they have been added to the code base. `NoIP` disables interpreter profiling; `BaseIP` uses the base implementation of IP which we already showed that it has high overhead. The names `NoLock`, `UnloadOpt`, `SkipBr`, `IPthread` and `DelayIP` correspond to techniques (a), (b), (d), (e) and (f) from the previous section. Each experiment adds a new technique on top of the existing ones.

The trends are similar on all benchmarks, but they can be best observed on Websphere Application Server startup due to their magnitude. While each implemented technique is useful to a certain degree, `ClassUnloadOpt` seems to come on top. The operations that are required to check whether the profiled data belongs to an unloaded class are rather expensive and they are executed quite frequently, therefore the important savings from eliminating most of the checks. Looking at Websphere Application Server startup, by eliminating the locks around the global hash table and minimizing the checks on unloaded classes we managed to reduce the startup overhead from 57% to 30%.

Skipping some of the branch records reduces the number of hash table accesses which we have seen as the most important source of overhead. This change improved Websphere Application Server startup time by another 7 percentage points.

Moving some of the operations on a dedicated IP thread does not reduce, but rather hides the overhead. While this technique proves effective here, additional processing power is required. On machines with one or two processors the additional IP thread might not help much.

Finally, the last technique reduces all three sources of IP overhead. By reducing the number of buffers that are collected by the VM, the JIT indirectly needs to parse fewer records and access the hash table less frequently. By applying this technique only during startup phase, we ensured that throughput does not degrade.

Figure 5 shows how throughput is affected by the techniques meant to reduce IP collection overhead. If we compare the bar entitled `BaseIP`,which implements none of the techniques, to the last bar in each group (`DelayIP`), which implements all the techniques, we see that the two are virtually identical (small differences are due to normal fluctuations in the performance numbers). This confirms that we achieved our goal of reducing IP overhead without affecting throughput at all.

To better support our explanations we also included some statistics related to the amount of data that requires processing. As shown in Table 4, as overhead is gradually removed, the number of buffers received for processing increases. The only exception is the last technique (`DelayIP`) where the VM avoids the filling of the buffers when the invocation count for methods is larger than 1000. The number of records scanned is proportional to the number of buffers, while the number of records processed takes a dip for the `SkipBr` technique which does not process some of the branch records.

The number of hash table writes (inserts or updates of the hash table data) equals the number of records processed and

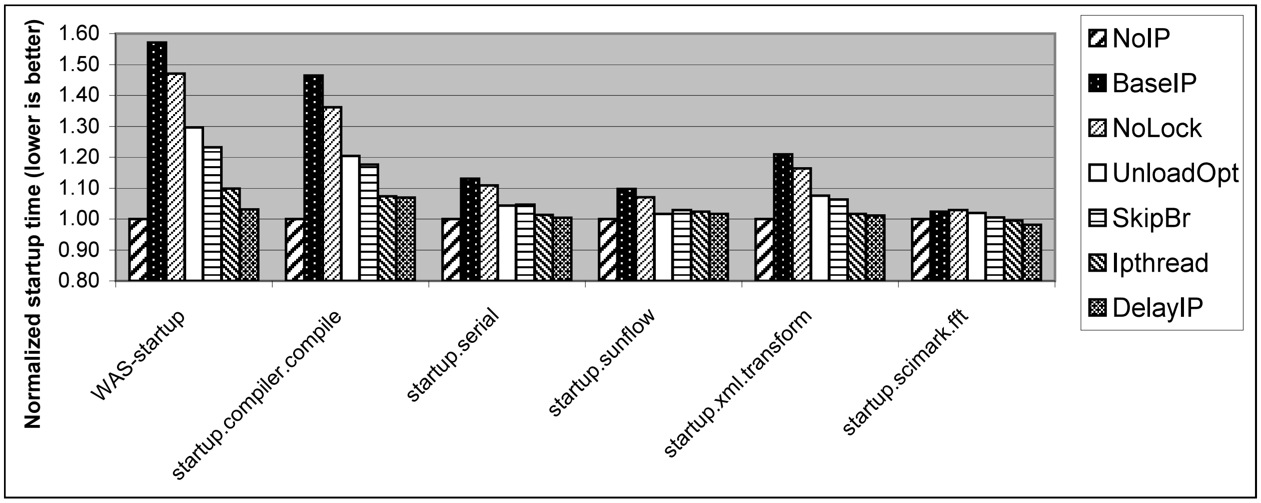

Figure 4. Effectiveness of described techniques on startup time.

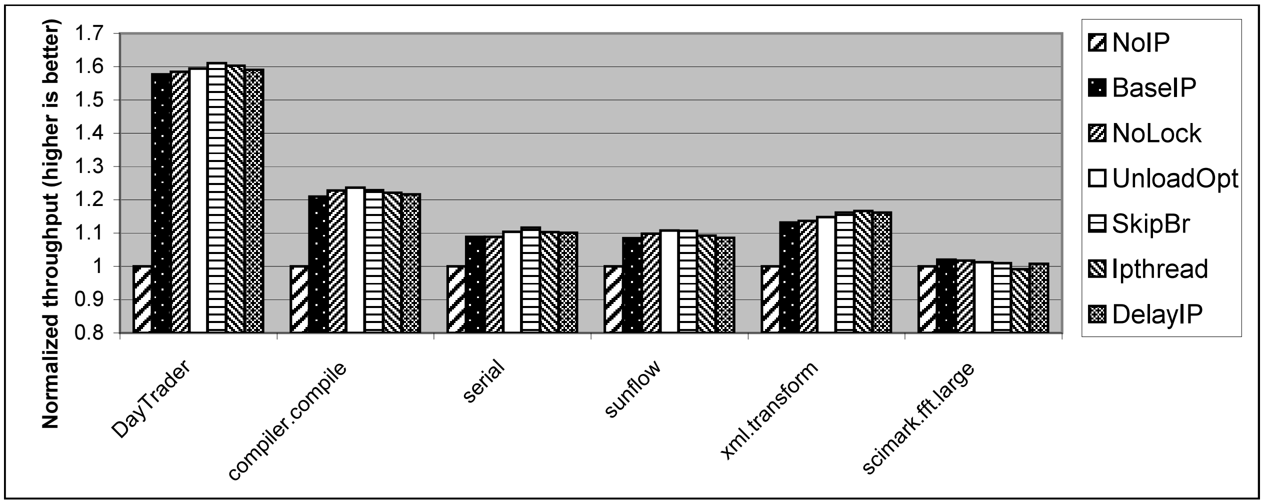

Figure 5. Effectiveness of described techniques on throughput.

	BaseIP	NoLock	UnlOpt	SkipBr	IPthread	DelayIP
Buffers received	394k	414k	472k	500k	555k	149k
Records scanned	33m	35m	40m	43m	47m	13m
Records processed	33m	35m	40m	28m	31m	9m
Hashtable reads	77k	71k	60k	57k	55k	51k
Hashtable writes	33m	35m	40m	28m	31m	9m

Table 4. Interpreter profiling statistics collected during the Websphere Application Server startup.

are in the millions. On the other hand, the number of hash table reads (generated by the JIT compiler when optimizing methods) are three orders of magnitude fewer. Thus, when attempting to reduce the IP overhead one should focus on streamlining the hash table access and reducing the amount of data to be processed.

6. Related work

There have been many papers that discuss profiling techniques where a second recompile is used with the previously generated profiling information. An example is the paper by Ball and Larus [10], which focuses on optimally placing counters to obtain accurate block execution frequency, yet reduce the information generated. Burger and Dybvig [11] have improved this technique and showed how to apply it to a dynamic compilation setting, where profiling information is collected on the fly and used in a recompilation infrastructure. In contrast, the profiling instrumentation described in this paper is implemented in the interpreter component. We have shown that instrumentation of dynamically generated code is impractical for a large number of methods because the overhead of instrumenting each method and recompiling can impact the application start-up and warm-up phase in a significant way. Our approach takes advantage of the execution properties of the interpreter component, and requires only one compilation to achieve the same result.

To reduce profiling overhead Arnold and Ryder propose a JIT profiling framework that collects profiling information in small bursts [1]. A profiling method has a replicated body: one version contains instrumented code, while the other contains the original code plus conditional checks pertaining on when to transition to the instrumented version. After sufficient profiling data has been collected, the method is recompiled. Hirzel and Chilimbi [2] have extended this framework by enlarging the size of the bursts and eliminating some of the checks. IBM's J9 VM implements a similar scheme, but in our experience this is still too heavyweight to use on enterprise-class applications.

Other papers focus on feedback-driven inlining. The technical report by Arnold and Sweeney from the IBM T.J. Watson research center [12] describes a technique based on periodical call-stack sampling, that is used to capture context sensitive call profiling information. A similar approach is described by Arnold and Grove in [13]. While sampling can successfully be used to control profiling overhead, for large applications this approach is too coarse-grained to collect enough data to be able to optimize a large number of callsites.

Upton et al. [14] present ways of reducing data collection overhead in the context of Pin [15]. The authors argue for a buffered collection approach and propose optimized code sequences to write data to the buffers, as well as mechanisms to reduce the cost of detection when the buffer becomes full. Our base IP implementation also uses a buffered collection approach. This was an early design choice based on our experience with JIT profiling and therefore it is not included in the mechanisms (a)-(f) described in Section 5. We have a totally different view on the importance of overhead sources. As we show in Section 4, with buffered collection in place the overhead of filling the buffers is not that important (4%). Instead, you should focus on managing the collected data in such a way as to become easily consumable by the JIT compilation threads (50% overhead).

7. Conclusion

In this paper we evaluated the efficiency and overhead of interpreter profiling in a production setting. We found interpreter profiling to be an invaluable mechanism in helping better optimize the Java code compiled at runtime with speedups ranging from 2% to 58%. However, contrary to the traditional wisdom, slowing down the interpreter does not go unnoticed. We observed that interpreter profiling has a profound effect on the startup time of applications (up to 57%). Larger applications with thousands of methods that must be compiled are more affected than smaller ones, mainly because the interpretation activity is more intense and the amount of collected data is bigger. Unfortunately it is exactly the larger applications where JIT profiling cannot be a substitute for interpreter profiling.

We also detailed some effective techniques that are used in IBM J9 VM to diminish or hide this overhead. The techniques presented here dramatically reduced the overhead and improved the startup time of applications. In the majority of the cases the remaining startup time regression due to IP was less than a few percent. We also ensured that the throughput does not get affected by a reduction in interpreter profiling data collected.

References

[1] M. Arnold, B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *PLDI* 2001.

[2] M. Hirzel and T. Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2001.

[3] V. Sundaresan, D. Maier, P. Ramarao, M. Stoodley. Experiences with Multithreading and Dynamic Class Loading in a Java Just-In-Time Compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[4] The Apache Software Foundation. Apache DayTrader Benchmark Sample. URL `https://cwiki.apache.org/GMOxDOC20/daytrader.html`

[5] Standard Performance Evaluation Corporation. SPECjvm2008. URL `http://www.spec.org/jvm2008`

[6] J. Valois. Lock-free linked lists using compare-and-swap. In *ACM Symposium on Principles of Distributed Computing*, 1995.

[7] T. Harris. A pragmatic implementation of non-blocking linked-lists. *Lecture Notes in Computer Science*, 2180:300-314, 2001.

[8] M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14-th annual ACM symposium on Parallel algorithms and architectures*, 2002.

[9] S. Heller, M. Herlih, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A Lazy Concurrent List-Based Set Algorithm. *Principles of Distributed Systems, Lecture Notes in Computer Science*, 3974:3-16, 2006

[10] T. Ball, J. R. Larus. Optimally Profiling and Tracing Programs. University of Wisconsin. ACM 1992.

[11] R. G. Burger, R. K. Dybvig. An Infrastructure for Profile-Driven Dynamic Recompilation. *Indiana University Computer Science Department*, 1997.

[12] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. *Technical Report RC 21789*, IBM T.J. Watson Research Center, July 2000.

[13] M. Arnold, D. Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[14] D. Upton, K. Hazelwood, R. Cohn and G. Lueck. Improving instrumentation speed via buffering. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, 2009.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapareddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, 2005.