Sampling profiler for Rotor as part of optimizing compilation system

Sofia Chilingarova
St-Petersburg State University
28, Universitetskiy pr.,
Petrodvorets
Russia 198504, St-Petersburg
sofie-chil@hotmail.ru

Vladimir O. Safonov
St-Petersburg State University
28, Universitetskiy pr.,
Petrodvorets
Russia 198504, St-Petersburg
v_o_safonov@mail.ru

ABSTRACT

This paper describes a low-overhead self-tuning sampling-based runtime profiler integrated into SSCLI virtual machine. Our profiler estimates how "hot" a method is and builds a call context graph based on managed stack samples analysis. The frequency of sampling is tuned dynamically at runtime, based on the information of how often the same activation record appears on top of the stack. The call graph is presented as a novel Call Context Map (CC-Map) structure that combines compact representation and accurate information about the context. It enables fast extraction of data helpful in making compilation decisions, as well as fast placing data into the map. Sampling mechanism is integrated with intrinsic Rotor mechanisms of thread preemption and stack walk. A separate system thread is responsible for organizing data in the CC-Map. This thread gathers and stores samples quickly queued by managed threads, thus decreasing the time they must hold up their user-scheduled job.

Keywords

SSCLI / Rotor, Just-in-time compilation, sampling-based profiling, de-virtualization, inlining.

1. INTRODUCTION

Optimization techniques based on profile data obtained at run time form the essential part of optimization strategy in modern dynamic compilation frameworks.[Arn02][Sug01][Jav02] Static analysis alone cannot provide sufficiently full information by sufficiently low cost to make optimizations pay for themselves in dynamic compilers. Managed environments have the distinguishing capability to provide feedback and use it in compilation at the very time the program executes, and runtime profilers are designed to utilize this capability. With profile data enabling selective optimization of the "hot" pieces of code we gain much more.

There are two main types of profile data optimizing compiler may be interested in: individual methods "hot counts", i.e. precise or approximate estimation of method execution frequency, and some kind of "call graph" which can provide information about the frequency of calls from one method to another. The former is used to pick up the individual "hot" methods for recompilation, the later helps to plan optimizations in the broader context taking into account the hot paths through the whole application.

Many techniques have been developed to collect and store runtime profile data. But the key point has always been a balance between the accuracy of profile data and low overhead of profiling facilities, which have to do their job at run time thus adding to compilation overhead. Experiment results show that strictly accurate profile is not necessary to make a good recompilation decision, so sampling profilers turned out an excellent tool to get rather complex information about program behavior with low overhead.

Typical sampling profiler working as a part of a dynamic compilation framework acts as follows: periodically it launches a task that looks up a stack for managed methods frames, then forms collected data into some structure reflecting dynamic call context and stores it for the subsequent use. [Arn02] [Wha00] Our profiler developed for SSCLI (Rotor) also utilizes this classical schema. It uses the mechanism for exploring stack that Rotor already has (we will cover it later) and stores data in Call Context

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

Map structure that contains counters for individual methods calls, total count for every call done by one method to the other, and detailed information about the context in which the call occurs.

Contributions

This paper makes the following contributions:

- **Data structures.** It describes a Call Context Map (CC-Map) data structure used for encoding runtime profile information. It shows the advantages of the Call Context Map: its capability to provide the full information needed for recompilation decisions quickly and with minimum effort remaining at the same time a rather compact structure. It describes the algorithm for filling CC-Map from a raw stack samples containing references to managed methods metadata and offsets in code.
- **Profiling Techniques.** The paper presents a profiling technique based on profiler and managed threads cooperation and background processing of raw samples data, which allows maintaining a complex structure of profile data storage not causing the managed threads to postpone their jobs for a long time. The bunch processing of samples helps to minimize synchronization on the global samples cache.
- **Experience using SSCLI features.** The paper shows how the SSCLI core functions and structures were used to help collecting stack samples and organizing profile data. It also describes the utilization of core SSCLI mechanism for threads cooperation and synchronization to provide cooperative behavior in gathering samples.
- **Evaluation of overhead and accuracy of profiling.** The paper presents evaluation of accuracy and overhead of the profiler ran on the SSCLI quality test suit using simple execution counters statistical correlation and Arnold & Ryder overlap percentage measure[Arn02].

2. RELATED WORK

Many papers published in the last years show the benefits of profile-driven optimizations and the ways profile data may be used in compilation decisions. Arnold[Arn02] in his PhD thesis paper describes in detail several kinds of profile-driven optimizations implemented in Jikes JVM. Suganuma et. al. [Sug01] in their review of IBM DK optimizing JIT-compilation framework give a full picture of how instrumentation and sampling based profiling is used to collect profile data from interpreted and compiled code, respectively. They provide experiment results showing the evident advantages of profile-based selective optimizing compilation over both optimizing non-selective and fast non-optimizing non-selective compilation.

Several studies show the practical use of dynamic profile data in such optimizations as inlining [Sug02] and devirtualization[Ish00]. These two types of optimization are very important for managed environments with intrinsic support of object-oriented languages where most method calls are virtual and many levels of indirection often present. Suganuma et. al. [Sug03] introduce an interesting optimization technique, Region-Based Compilation, that allows more effective use of profile data.

Whaley[Wha00] describes several different approaches to profile data organization: Dynamic Call Graph (DCG), Calling Context Tree (CCT), Partial Calling Context Tree (PCCT). Arnold et. al. [Arn00] shows in more detail how the DCG is constructed. We'll look closer at these structures in the next section where we describe our data representation choice, Call Context Map (CC-Map), and compare it with the other options. CC-Map is in many respects similar to CCT and PCCT, but provides easier ways to retrieve full context information. Also we don't place such restrictions on the length of a sample, as PCCT-based approach described by Whaley. In our profiling framework we allow sample buffers to grow when needed, although we define some rather high limit for the cases of incredibly deep stack, which are rare.

Arnold and Grove [Arn05] propose an interesting variation of samples collection technique. Instead of taking one sample at a time, their profiler takes a bunch of samples: when profiling is requested, stack walk is performed several times over a short interval. Authors show how this approach helps eliminate inaccuracy in some situations.

3. PROFILER DESIGN

In this section we describe an overall structure of the profiler: how the sample data storage is organized and how the samples gathering mechanism works. We introduce a Call Context Map (CC-Map) that allows easy retrieving of many kinds of data needed for compilation/recompilation decisions. We present a sampling strategy that helps to maintain a rather complex CC-Map structure and at the same time not cause the user threads job to be postponed for long intervals. In the next section we'll take a closer look at the Rotor-specific issues and show how the profiler uses intrinsic mechanisms of the SSCLI virtual machine to do its job.

Call Context Map

3.1.1 Previous approaches

The common way to represent the sequences of calls with their relative frequency in runtime profile data is using some kind of call context tree. Call context tree consists of nodes correspondent to the method calls and directed edges, which denote caller-callee relations. The examples are Dynamic Call Graphs (DCG, DCG-E) described by Arnold et. al. [Arn00] and Calling Context Tree/Partial Calling Context Tree (CCT, PCCT) described by Whaley[Wha00] . Dynamic Call Graph is shown on the Figure 1b. Nodes represent method calls, edges mark associations between caller and callee, and weights assigned to edges mean the number of calls from the specified caller to the specified callee encountered in samples. This is rather compact representation but the information we can retrieve from it is limited. We can estimate how often one method calls the other, but with DCG alone we cannot determine, for example, that call chain ACD has never been encountered in samples, ABC has been encountered 2 times, and BCD – only once. Thus DCG can effectively represent only one-level-depth profile.

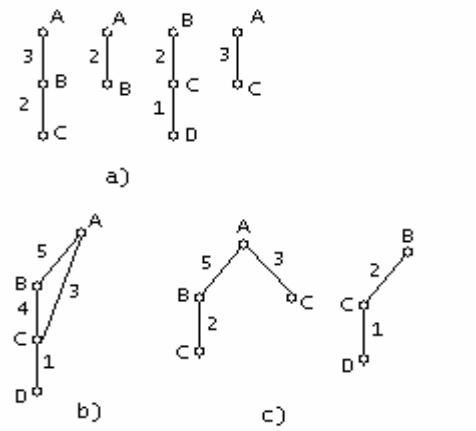

Figure 1. DCG and PCCT structures: a) samples collected from stack; b) correspondent DCG; c) correspondent PCCT

Partial Calling Context Trees (CCT) shown on Figure 1c provides more context information. Details of PCCT construction are covered in [Wha00]. They build PCCT using the fixed length buffer for samples, so that a delay does not be very long when the stack is extremely deep. When a sample is got and a PCC-Tree with the outer caller as a root is found, profiler updates counters for edges in this tree, otherwise a new tree is created. Here we can point out longer call sequences, but still cannot know, without additional analysis, that calls from B to C have been encountered 4 times, totally. To retrieve this information we should examine all the trees looking for edges BC and adding the counters to the total sum.

One more problem is illustrated by Figure 2a. Let we have a call graph shown at the left side of the figure. A and E call B and in both cases B calls C. Then C calls D or F. Also the samples with B as the outer frame are found, as shown on the figure. Let we build the Call Context Trees form these samples. We get three of them, with A, E, and B as roots.

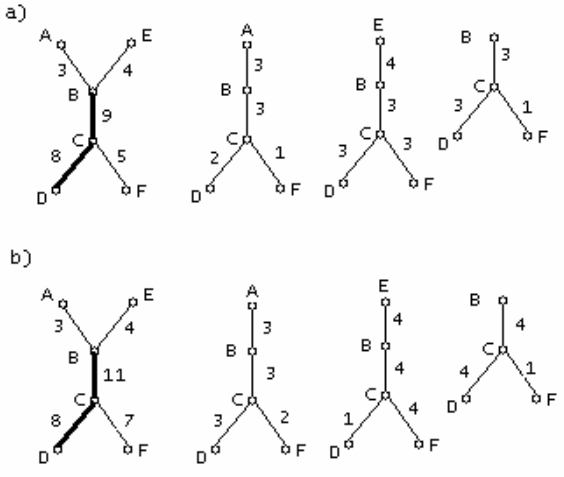

Figure 2. More complex call context

Here the hottest path is actually BCD, which executes 8 times. But we cannot retrieve this information automatically having only the CC-trees in hand. We cannot queue BCD path for possible recompilation automatically when the total counter exceeds threshold because we haven's such a total counter. The solution might be to construct/update CCT for every caller in the chain when a sample is got, but this way we fail to distinguish the frequencies of call to BCD in different contexts. For example, if the situation is like the one shown on Figure 2b, we'll fail to know that BCD path (executes 8 times totally) is actual only for calls from A. For E call site the path EBCF is really hot. The PCC-trees for this case (3 trees shown at the right side of the Figure 2b) reveal it clearly. If we update counters for BC and CD in the tree with B root every time the path is encountered in a sample, at any place, we capture the information about the total number of execution of BCD, but loose the important context information. So we need some combination of the described approaches.

3.1.2 Call Context Map Structure

Call Context Map (CC-Map) structure is designed to address issues depicted in the previous subsection. The higher level of the CC-Map is a hash-table containing references to *MethodProfile* nodes. *MethodProfile* node stores a total counter for the method executions and references to the nodes

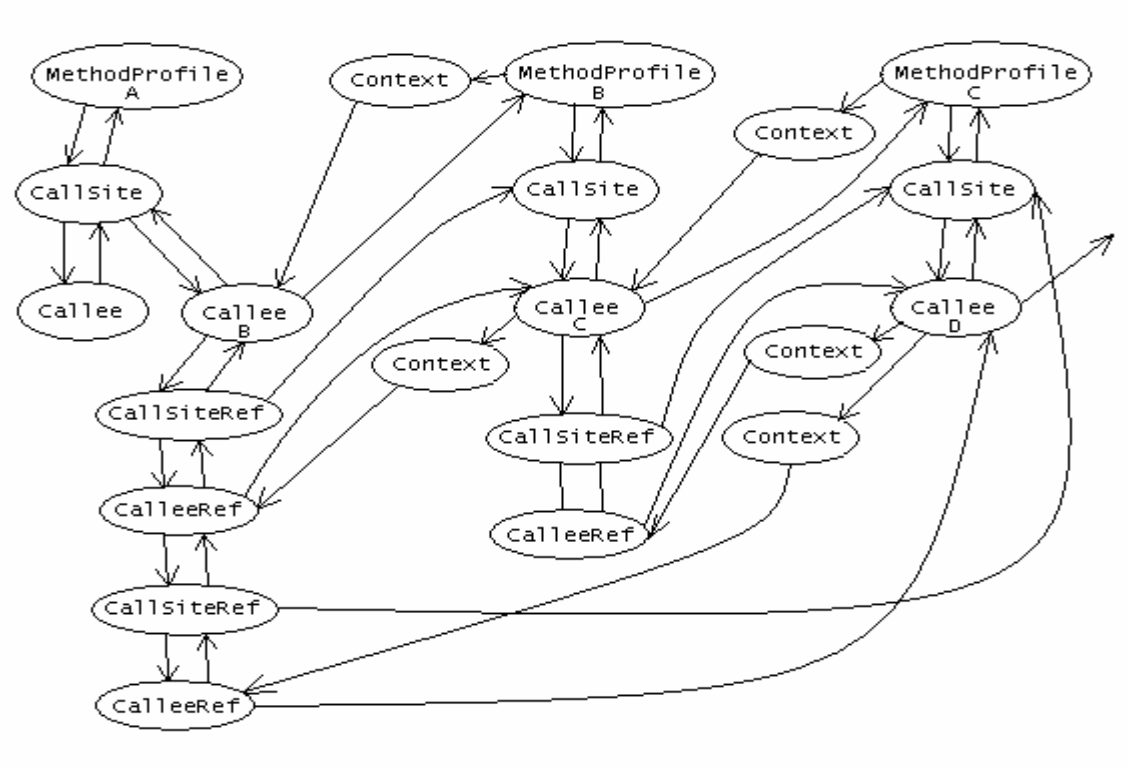

Figure 3. Call Context Map fragment

representing information about calls from this method to the others.

The *Callee* nodes contain accumulated counters for the total number of calls from the concrete caller to the concrete callee, in any context. Additionally, the tree of reference nodes is constructed for every call sequence. These *Ref* nodes contain counters for calls done in the given context and references to the nodes, which store general information about the call.

A fragment of CC-Map structure is shown on Figure 3. Let method A calls method B, B calls C, and C calls D. Every caller profile refers to *CallSite* node that contains general information about the call site – offset, reference to the caller profile, etc. *CallSite* node refers to one or more *Callee* nodes, which store call counters and, in turn, refer to the profiles of callees. *CallSiteRef* and *CalleeRef* nodes refer to the general *CallSite* and *Callee* nodes and *CalleeRef* nodes store the context counters. Every node representing general call information has *Context* references to the nodes, which describe a context of the call.

3.1.3 Advantages of CC-Map structure

CC-Map accumulates a total call count for every caller-callee pair and at the same time it allows retrieving information concerning calls in the specific context. This information is easily available: a compilation controller may lookup contexts by the *Context* references when some counter exceeds a threshold, as well as move up and down through a call chain.

From the *CallSite* and *CallSiteRef* nodes a controller can know whether the call has probably one target (and so consider devirtualization). *CallSite* node provides this information for all calls from a given site, *CallSiteRef* – only for calls done in a given context.

CC-Map is a rather compact structure. Nodes don't store duplicate data. CC-Map allows quick updating, as well as rather quick removing of nodes, which appear cold. Compilation controller need not perform additional analysis of trees to get information necessary for good decision: it can only follow references.

Figure 4 shows an example: a simplified view of CC-Map for the calling sequences presented on Figure 2a and 2b. The *CallSite* nodes are omitted for simplicity, as there is only one call site for each method in this example. You can see that a bi-directional association exists between a node with general information about method call and nodes representing the same call in the different contexts. When an event of a total counter exceeding threshold takes place, a compilation/recompilation controller can quickly look through the contexts to make an appropriate

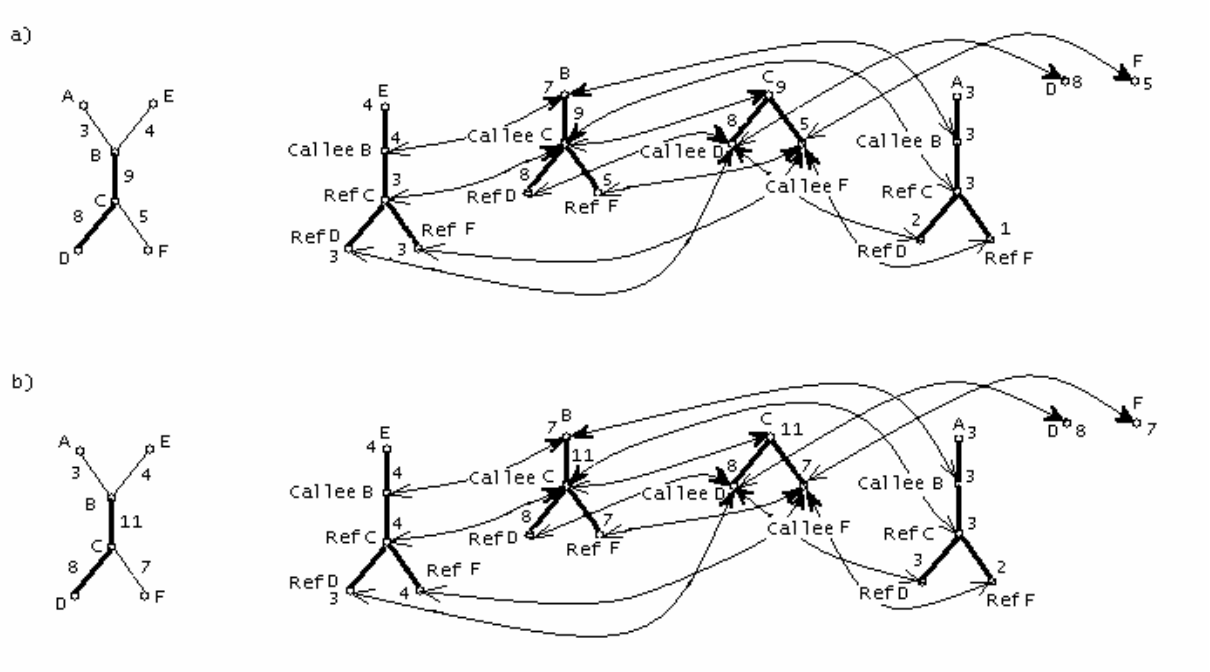

Figure 4. CC-Map for Fig. 2 examples. Bold arrows indicate references from nodes describing call in a given context, thin arrows indicate references form a general information node to call-in-context nodes (this association is represented by "Context" items on Fig. 3). The roots of the trees are MethodProfile nodes containing the total counters for method executions

compilation decision (for example, consider the common callers for de-virtualization or inlining too, especially if only one callee has been detected at the correspondent call sites so far). When analyzing a frequently executed call sequence a controller can browse all general call information nodes and access other contexts from them. It can move up and down the call sequence representation (see Fig. 3) to gather all the information about callers and callees that might affect a recompilation strategy choice.

3.1.4 CC-Map filling and updating

When a sample is being taken, all the data initially is written into a buffer. The stack lookup starts from the top of the stack and ends at the outermost frame or at the first managed method activation record that has already been visited by profiler. The profiler marks managed method activation records when looks them up (the JIT-compiler is configured to push the additional slot on the stack for this purpose), so during the following passes it can distinguish the new frames from the old ones. When the profiler encounters an old (marked as already visited) frame, it records this frame data (as it is needed to register a new call from the frame) and stops looking up the stack.

So, at the start of the buffer we have a reference to the method correspondent to the activation record at the top of the stack (i.e., most inner call), and at the end of the buffer – the outer caller (or the innermost call that hasn't returned from the previous lookup) reference.

The pseudocode for sample buffer processing looks as follows:

```
For(int i = 0; i < end_of_sample; i++)
{
   update MethodProfile(buf[i]);
   if (i > 0)
   {
      update Callee(buf[i],buf[i-1]);
   }
   for (j = i-2; j>=0;j--)
   {
      update CalleeRef(buf[j]);
   }
}
```

The real code is a little more optimized and a little more complicated, but the underlying algorithm is the same.

Profiling Algorithm

Maintaining such a complex structure as the CC-Map requires some effort. Algorithm described in the previous section may take a long time to complete. But we cannot afford to stop user threads for observable intervals because of profiling.

The solution we have chosen is to separate taking sample from thread stack from storing the sample data in the CC-Map. For this purpose we use two profiler worker threads, as well as thread-local and

global queues for samples waiting for the profiler to process them.

Profiling job is launched by the *MarkThreadsWorker* system thread which marks every live managed thread to make it know that it should take a sample when reaches a safe point. Every live managed thread has its own sample buffer and its own short samples queue. The sample is written into the thread local buffer and pushed into the thread local queue. When local queue length exceeds a threshold (rather low, now 10) all its contents is pushed to the global queue. This schema is aimed to decrease the need to grab a global queue lock, and thus to decrease possible pauses caused by waiting for the lock. Little delay in samples processing is not critical because only large numbers are considered when making compilation decisions.

The CC-Map manager thread periodically grabs the global queue lock, takes out a bunch of samples and put them into its own queue. Then it releases the lock and proceeds with processing samples without hurry. Global queue hashes samples by thread id so the CC-Map manager thread can return the processed sample buffers back to their thread so that it need not to allocate new memory. Local thread buffer grows automatically when needed, queued samples buffers grow then they need to adapt to local buffer size. So when threads get back their own buffers, previously queued, these buffers are likely to have appropriate size. If the thread is already finished when CC-Map manager returns processed sample buffers for it, this chain of buffers is put aside to be used by next new thread.

Tuning Sampling Interval

The profiler is, *self-tuning*, it adapts an interval of taking samples to the characteristics of environment where it runs. To do this it uses a simple heuristics: it tracks how often the same activation records appear on the top of the stack. It doesn't take much effort or time: as the profiler already distinguishes between visited and not visited frames and stops at the first visited, we need only to reflect this condition in a sample and check whether this frame is the first in a sample (i.e. it is taken from the top of the stack) when processing the sample. If so, a special counter is incremented.

There are two threshold values defined: maximum percentage of repetitions and minimum percentage of repetitions. CC-Map manager thread evaluates actual percentage of repetitions (of activation record appearance on the top of the stack) every 1000 samples (more precisely, than processed samples portions is more than 1000, because the manager thread handles a bunch of samples in every pass). If percentage of repetitions is lower then minimum threshold, it is considered too low and sampling interval decreases. If percentage of repetitions is higher than maximum threshold, the sampling interval increases.

4. INTEGRATION WITH ROTOR

Rotor has a built-in mechanism for walking the stack, which is used for such purposes as exception handling and security checks[Stu03]. It involves several methods and functions of virtual machine and among them the *StackWalkFrames* method of the VM *Thread* class, which we use to take samples. *StackWalkFrames* takes a function to execute on every encountered stack frame as a parameter, so its work is easily customizable. The advantage of using it is that it already knows how to distinguish managed method frames from unmanaged method frames, can recognize context transitions (e.g. across application domain boundaries), encapsulates calls to Rotor facilities to get metadata references and offsets, and it provides a convenient interface to do jobs on the stack.

We make managed threads call *StackWalkFrames* method at, so called, "safe points", building upon the other intrinsic Rotor mechanism – trapping threads when they know that it is safe to suspend now. This mechanism has been originally used to trigger garbage collection. Checks for a suspension request have been inserted by the JIT-compiler at back edges and everywhere where the next piece of code may take long time to execute[Stu03]. Such checks are also performed by some of runtime helper functions extensively used in Rotor. We utilize this mechanism and add additional check points at the entry of every method. At that new check points we test only for the need to take sample.

We also used the SSCLI core HashMap class to construct the CC-Map in Rotor. SSCLI HashMap class implements a hash table used by VM for its internal needs. It hashes pointer type values by the pointer type keys (so allows storing profile objects by the pointer-to-metadata keys), implements locking for insert, delete and lookup, and takes care of cleaning up itself. It is just what we need. So we choose HashMap as a hash table to store MethodProfile references at the highest level of CC-Map and as a hash table to hold queues of samples waiting for processing in the global samples store.

5. RESULTS

We tested our profiler on SSCLI 1.0. To measure overhead and accuracy of profiling we used tests from a suite supplied with SSCLI. To estimate overhead we chose a set of base tests from *bcl\system*

and *bvt* subdirectories and tests from *bcl\threadsafety* subdirectory of Rotor *tests* directory. To estimate accuracy we used tests from *bcl\threadsafety* subdirectory, where multiple threads execute the same code. As measures we used statistical correlation of the total executions counters stored in *MethodProfile* nodes and Arnold & Ryder overlap percentage[Arn02] for the whole tree comparison. Overlap percentage of trees T1 and T2 is computed as follows:

$\sum_{N \text{ in } T1,T2}$ [min (Weight (N_{T1}), Weight(N_{T2})]

where Weight (N_{Tx}) is:

value(N_{Tx})/$\sum_{N \text{ in } Tx}$value(N),

N is a node holding a counter, value is a value of the counter. When N is not found in Tx (thought it exist in Ty and thus in TxTy set), it is assumed that value(N_{Tx}) = 0.

For performance test the low threshold for repetitions (cases when the same method appears on the top of the stack) was set to 1%, high threshold for repetitions was set to 15%. For the correlation and overlap measurement tests the self-tuning was turned off, because it can affect the correlation results distinctly for short-running tests, as those we used. However the great deal of these differences is produced at the interval when the profiler is tuning, so such results do not reflect the real picture in steady state. Logging of sample interval changes in the process of tuning revealed that the sample interval becomes stable after 1-2 changes. We measured correlation and tree overlap with different sample intervals (with self-tuning turned off) and the best results (95-99%) were obtained with the same interval that the profiler found automatically.

In accuracy test we recorded and compared executions counters and the whole CC-Maps from 10 subsequent runs. The results of every run were compared with results of every other and an average value was computed.

To make the CC-Map accessible even after the VM was stopped running, we dumped the CC-Map (in the *fastchecked* mode) to an XML file at VM shutdown. Then original CC-Maps were restored from XML representation and compared (in XML dump of CC-Map managed methods are identified by the full name and signature to make comparison possible, though at runtime they identified only by pointer to metadata).

Table 1 shows the average correlation for 10 subsequent runs of the same test and average tree overlap percentage. All the tests are from *bcl\threadsafety* suite.

Test Name	Correlation, %	Overlap, %
co8545int32	99	97
co8546int16	99	92
co8547sbyte	99	94
co8548intptr	99	98
co8549uint16	99	95
co8550uint32	99	95
co8551byte	99	97
co8552uintptr	99	97
co8553char	99	96
co8555boolean	99	96
co8559enum	98	75
co8788stringbuilder	99	67
co8827console	99	77
co8830single	99	98

Table 1. Average correlation for total executions counters and overlap percentage extracted from comparison of results of 10 subsequent runs

We can see that though the correlation of simple execution counters is always good (98-99%), overlap percentage sometimes appears lower than 80%. We think, however, this can be probably explained by the fact than the tests themselves were very short.

Tests were run on Celeron433 processor, 256M RAM. Sampling interval was set to 10ms. This is rather short interval for this hardware configuration and for long-running programs in may be longer. However, the tuning mechanism can adjust the interval well. When testing we started from interval 50ms, and for the tests, which performed bad with such an interval, the profiler made it less. For the tests, which performed well, the interval remained unchanged. We see also in Table 1, that for some tests accuracy is even redundant. 95-97% would be enough to consider results statistically significant. For the cases when we can get such accuracy with longer interval, it will not decrease (or it can even increase if the initial interval appears too short).

The profiling overhead was measured on the free build against unchanged Rotor free build, on the same hardware configuration, on the tests from *bcl\system, bvt*, and *bcl\threadsafety* subsets of Rotor core test suit. Initial sampling interval was set to 50ms. Tuning was turned on. Tests were run 2 times, and the total overhead did not exceed 3%. In the future we intend to consider automatic turning off tuning after a certain period of time so that to lower overhead.

6. REFERENCES

[Arn00] Arnold, M., Fink, S., Sarkar, V., Sweeney, P. A comparative study of static and dynamic heuristics for inlining. In ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, Jan. 2000.

[Arn02] Arnold, M. Online Profiling and Feedback-Directed Optimization of Java. PhD thesis, Rutgers University, October 2002.

[Arn05] Arnold, M. and Grove, D. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In Proceedings of the international Symposium on Code Generation and Optimization, March 20 - 23, 2005.

[Jav02] The Java HotSpot™ Virtual Machine, v1.4.1, d2, A Technical White Paper. Sun Microsystems, September 2002.

[Ish00] Ishizaki, K., Kawahito, M., Yasue T., Nakatani, T. A study of devirtualization techniques for a Java just-in-time compiler. In ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Oct. 2000.

[Stu03] Stutz, D., Neward, T., Shilling, G. Shared Source CLI Essentials. O'Reilly, 2003.

[Sug01] Suganuma, T., Yasue, T., Kawahito, M., Komatsu, H., Nakatani, T. A dynamic optimization framework for a Java just-in-time compiler. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 2001.

[Sug02] Suganuma, T., Yasue, T., Nakatani, T.: An empirical study of method inlining for a Java Just-In-Time compiler. In: Proceedings of USENIX 2nd Java Virtual Machine Research and Technology Symposium (JVM'02), pp. 91–104, 2002.

[Sug03] Suganuma, T., Yasue, T., Nakatani, T., A Region-Based Compilation Technique for a Java Just-In-Time Compiler, ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003), pp. 312-323, June 9-11, 2003.

[Wha00] Whaley, J. A portable sampling-based profiler for Java virtual machines. In ACM 2000 Java Grande Conference, June 2000.